UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR


GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE GRADO

# PROPUESTA DE SOLUCIÓN LOGÍSTICA EN REDES DE TRANSPORTE.

AUTOR: Ernesto Carvajal Lastres

TUTOR: María Dolores Cuadra Fernández

Leganés, 20 de septiembre 2013

# Índice de contenidos


1 Introducción ..... 5

1.1 Motivación y objetivos ..... 6

1.2 Estructura del documento ..... 6

2 Definición del problema ..... 7

2.1 Notación básica ..... 8

2.2 Requisitos y restricciones ..... 9

3 Estado del arte ..... 9

3.1 Métodos de solución exacta ..... 10

3.1.1 Búsqueda exhaustiva ..... 10

3.1.2 Branch and bound ..... 11

3.1.3 Programación lineal ..... 14

3.2 Heurísticas clásicas ..... 16

3.2.1 Algoritmo de Clarke and Wright ..... 17

3.2.2 Algoritmo de barrido ..... 18

3.3 Meta-heurísticas ..... 20

3.3.1 Simulated annealing ..... 21

3.3.2 Tabu search ..... 22

4 Diseño y desarrollo de la solución ..... 24

4.1 Requisitos del problema a resolver ..... 24

4.2 Aspectos técnicos ..... 26

4.3 Solución inicial mediante Clarke and Wright ..... 30

4.4 Optimización mediante Branch and Bound ..... 32

4.5 Soluciones sucesivas mediante Taburoute ..... 34

4.5.1 Preámbulo y notación de Taburoute ..... 35

4.5.2 Pseudocódigo del procedimiento Search(P) ..... 37

4.5.3 Pseudocódigo del Algoritmo Taburoute ..... 38

4.5.4 Selección de parámetros ..... 38
4.5.5 Inserción de vértices mediante Stringing ..... 39
4.5.6 Eliminación de vértices mediante Unstringing ..... 41
4.5.7 Pseudocódigo de GENI ..... 42
4.5.8 Pseudocódigo de US ..... 42
4.5.9 Resultado de Taburoute ..... 42
5 Resultados y evaluación ..... 43
6 Posibles mejoras ..... 47
6.1 Problema de escalabilidad y solución ..... 47
6.2 Problemas de versatilidad y soluciones ..... 48
7 Planificación y costes ..... 49
8 Conclusiones ..... 50
9 Bibliografía ..... 52

# Índice de ilustraciones

Ilustración 1.1 Representación gráfica del VRP ..... 5
Ilustración 2.1 Versiones populares del VRP y sus interrelaciones ..... 8
Ilustración 3.1 Árbol de estados. ..... 12
Ilustración 3.2 Grafo inválido que cumple las primeras restricciones ..... 15
Ilustración 3.3 Representación gráfica del concepto de saving ..... 17
Ilustración 3.4 Representación gráfica del Algoritmo de barrido ..... 20
Ilustración 4.1 Interfaz gráfica de pruebas ..... 27
Ilustración 4.2 Organización en proyectos de la solución ..... 28
Ilustración 4.3 Adaptación del ejemplo introductorio ..... 31
Ilustración 4.4 Adaptación del concepto de saving ..... 31
Ilustración 4.5 Rutas calculadas por Clarke and Wright ..... 32
Ilustración 4.6 Árbol cadena ..... 33

Ilustración 4.7 Optimización de rutas mediante Branch and bound ..... 34
Ilustración 4.8 Stringing tipo 1..... 40
Ilustración 4.9 Stringing tipo 2..... 40
Ilustración 4.10 Unstringing tipo 1 ..... 41
Ilustración 4.11 Unstringing tipo 2 ..... 41
Ilustración 4.12 Solución calculada por Taburoute ..... 43
Ilustración 5.1 Solución calculada para el caso 1 ..... 44
Ilustración 5.2 Solución calculada para el caso 2 ..... 44
Ilustración 5.3 Solución calculada para el caso 3 ..... 45
Ilustración 5.4 Solución calculada para el caso 4 ..... 45
Ilustración 5.5 Solución calculada para el caso 5 ..... 46
Ilustración 5.6 Solución calculada para el caso 6 ..... 46
Ilustración 8.1 Evolución de las heurísticas para el VRP ..... 51

# Índice de tablas

Tabla 3.1 Resultados experimentales de Clarke and Wright ..... 18
Tabla 3.2 Resultados experimentales de simulated annaeling ..... 22
Tabla 3.3 Resultados experimentales de Taburoute ..... 24
Tabla 5.1 Tabla comparativa de los casos de prueba ..... 47
Tabla 7.1 Diagrama de Gantt con las fases del proyecto ..... 50
Tabla 7.2 Características de cada fase del proyecto ..... 50

# 1 INTRODUCCIÓN

Durante las últimas décadas ha incrementado considerablemente el uso de técnicas de *Investigación Operativa* y *Optimización Matemática* en paquetes comerciales con el objetivo de gestionar de manera eficiente bienes y servicios. El amplio uso de estas técnicas, tanto en Europa como en Norte América, ha mostrado recortes en los costos de transportación global entre el 5% y el 20%. El éxito de estas técnicas es consecuencia directa de la explosión digital que trae consigo la creciente integración de los sistemas de información con los procesos productivos y comerciales.

Se han propuesto decenas de problemas de optimización, cada uno con su propia importancia. Este documento se enfoca en el problema de distribución o recogida de bienes (o personas) desde sus ubicaciones iniciales hasta un destino final. Este problema se conoce como Problema de Enrutamiento de Vehículos (VRP, por sus siglas en inglés).

El VRP consiste en encontrar el conjunto de rutas óptimas a ejecutar por una flota de vehículos para ofrecer un determinado servicio a uno o más clientes. Es fácil observar que es uno de los problemas más importantes y minuciosamente estudiados en el campo de la *Optimización Combinatorial*.

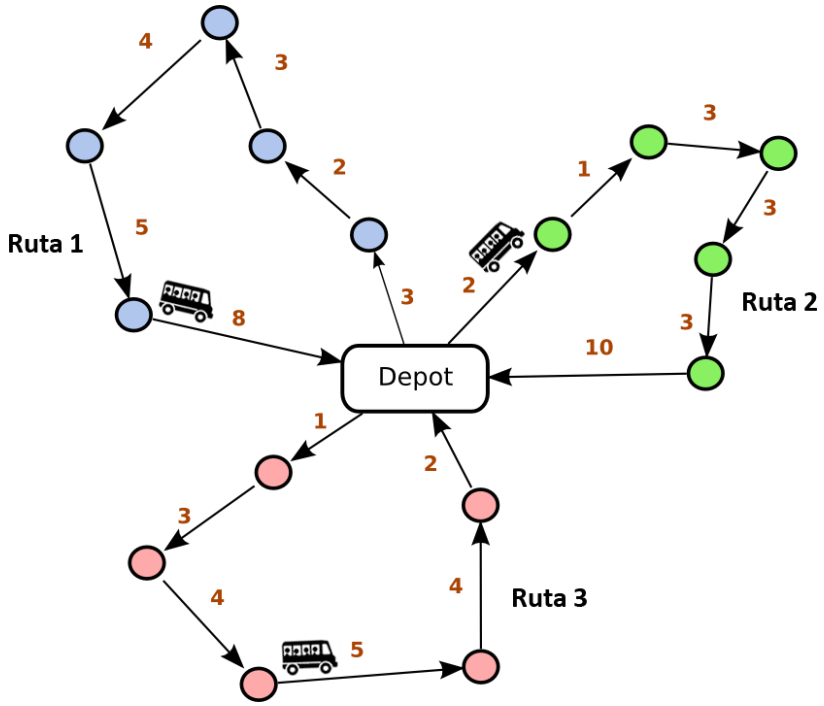


ILUSTRACIÓN 1.1 REPRESENTACIÓN GRÁFICA DEL VRP

Ha pasado más de medio siglo desde que Dantzig y Ramser propusieron el problema por primera vez en 1959. Por aquel entonces estos trataban de modelar la solución a un problema real que involucraba el reparto de combustible por diferentes estaciones. Dicha necesidad inspiró la primera formalización matemática del problema así como la primera aproximación algorítmica. Media década más tarde Clarke y Wright propusieron

una simple estrategia *greedy* que mejoraba considerablemente la aproximación inicial. Estos dos papers iniciales fueron la semilla que germinara cientos de modelos y aproximaciones algorítmicas, tanto para ofrecer soluciones óptimas como aproximadas, a las diferentes versiones del VRP. Docenas de empresas en el mundo trabajan para fabricar soluciones comerciales a las múltiples versiones de este problema que aparecen en el mundo real. El gran interés que rodea al VRP se debe a dos cosas: su relevancia práctica y su considerable dificultad. Actualmente las instancias más grandes a las que se puede encontrar solución óptima (sin tener en cuenta particularidades de la entrada) rondan los 100 clientes.

## 1.1 MOTIVACIÓN Y OBJETIVOS

GoalSystems es una empresa madrileña especializada en el diseño e implantación de sistemas que utilizan técnicas avanzadas de optimización para mejorar la rentabilidad a través de la planificación de transporte, personal y recursos en general. Entre sus clientes más importantes está Iberia, que utiliza GoalStaff (un sistema desarrollado por GoalSystems) para prever el costo asociado al crecimiento y sobre todo para coordinar y organizar a 12.000 trabajadores, de forma eficiente y automática. Otro de sus proyectos estrellas fue restructurar el transporte urbano de una ciudad de cerca de 8 millones de habitantes, con la empresa TransMilenio en Bogotá. En enero del 2013 me incorporé a GoalSystems como becario.

Para la fecha de mi incorporación varios clientes de la empresa estaban solicitando el desarrollo de un sistema que calculara rutas óptimas para la recogida de personal y productos. Unos cuatro clientes estaban solicitando en resumen una solución comercial al VRP variaciones específicas en cada caso. Hasta este momento la empresa no había invertido recursos en investigar ni diseñar una solución a este problema.

En GoalSystems saben cómo darle la bienvenida a un becario, y pasadas 4 horas de mi primera jornada, mi tutor de prácticas me puso al tanto de la situación y me informó que durante los próximos meses estaría investigando, diseñando e implementando una solución al VRP. Aunque un poco abrumadora, la noticia me llenó de ansiedad, ya tendría la oportunidad de investigar a fondo el estado del arte de un problema bien conocido, para adaptar posteriormente la teoría a nuestras necesidades prácticas y diseñar e implementar una solución.

## 1.2 ESTRUCTURA DEL DOCUMENTO

Este documento está dividido en dos partes.

La primera parte tiene un propósito teórico. Se introduce el problema formalmente, así como sus diferentes variaciones. Además serán explicados brevemente algoritmos de solución al VRP que se han considerado los más relevantes de su tipo. Los algoritmos han sido agrupados en familias dependiendo de la estrategia que siguen: “Métodos de solución exacta”, “Heurísticas clásicas” y “Meta-heurísticas”. Esta agrupación además

coincide convenientemente con una agrupación cronológica. En esto se abundará más adelante.

La segunda parte hace referencia a la práctica. Se explica qué algoritmos se han seleccionado y por qué, así como la adaptación de dichos algoritmos a las particularidades de nuestro problema. Además se cubren brevemente varios aspectos técnicos del desarrollo, como entorno utilizado, lenguaje, características del entorno de prueba, etc. Se ha dedicado una sección para ilustrar los resultados prácticos del sistema implementado y sus tiempos de cálculo para instancias específicas. Finalmente se comentan problemas de escalabilidad y otras debilidades identificadas, así como posibles mejoras futuras.

La redacción de este documento asume que el lector tiene conocimientos básicos de simbología matemática, lógica y notación de conjuntos. Además se ha asumido que el lector maneja términos de teoría de grafos y conoce los algoritmos más importantes que resuelven problemas fundamentales de este campo (árbol de recubrimiento minimal, problema del camino más corto, etc.).

# 2 DEFINICIÓN DEL PROBLEMA

En esta sección se da una definición formal del problema, modelado en el campo de la teoría de grafos. Además será introducida la notación relevante y la terminología utilizada durante el resto del documento. Es importante comentar que el problema detallado en esta sección se corresponde con el VRP "puro", tal cual aparece en la bibliografía. Especificidades del problema a resolver y adaptaciones de los algoritmos serán debatidas más adelante.

Las necesidades del mundo real, dinámico y cambiante, han obligado a los estudiosos del VRP a enfocarse no solo en un problema, sino en una familia de problemas que se relacionan entre sí. Los problemas de esta familia se diferencian unos de los otros en las restricciones que consideran. Mientras más restricciones, más complejidad. Cada uno de estos problemas recibe un código de letras que está directamente relacionado con el nombre completo en inglés del problema en cuestión. La Ilustración 2.1 resume cuales son los principales problemas estudiados y la relación entre ellos. Una flecha que sale del problema A y llega al B significa que el problema B es una extensión del problema A, o sea, que se le han agregado restricciones.

Para lograr brevedad, en este documento nos vamos a enfocar solo en el problema de enrutamiento de vehículos con restricciones de capacidad máxima de cada vehículo y distancia máxima de cada ruta (DCVRP), ya que cómo veremos posteriormente es el que más se ajusta al problema que queremos resolver.

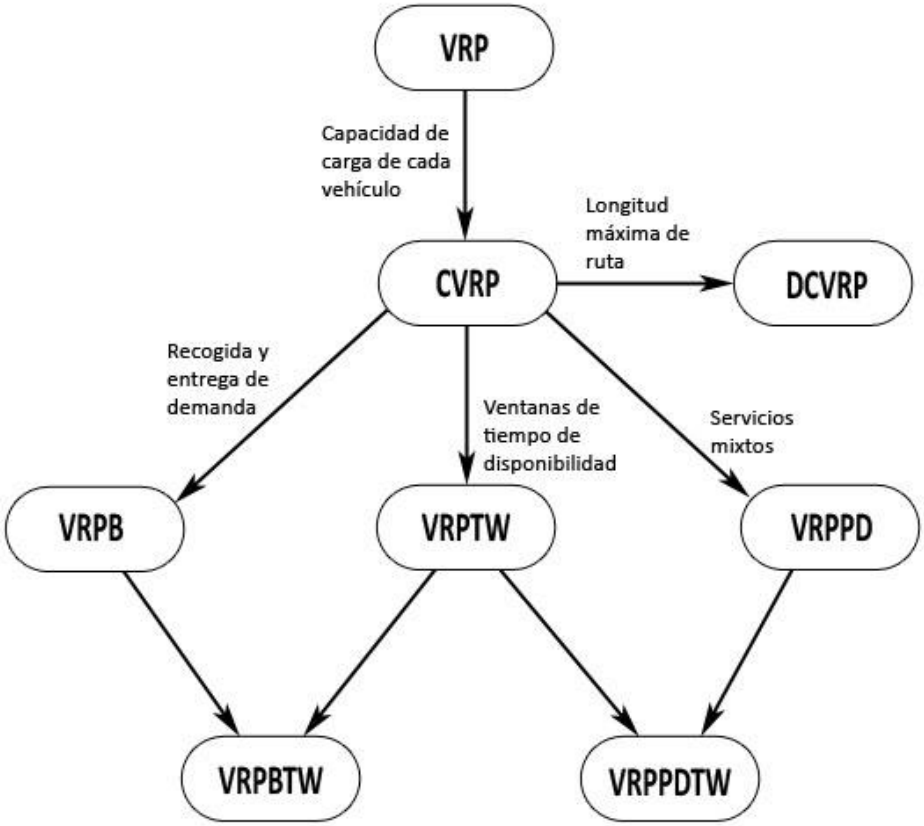


ILUSTRACIÓN 2.1 VERSIONES POPULARES DEL VRP Y SUS INTERRELACIONES

## 2.1 NOTACIÓN BÁSICA

El DCVRP puede ser descrito como el siguiente problema teórico de grafos. Sea $G = (V, A)$ un grafo *completo* donde $V = (0, \dots, n)$ es el conjunto de los vértices y $A$ es el conjunto de los arcos. Los vértices $i = 1, \dots, n$ corresponden a los clientes mientras el vértice $0$ es el destino (de ahora en adelante llamado *depot*). Algunas veces el *depot* es asociado con el vértice $n + 1$.

Una matriz de costos no negativos $c_{ij}$ , está asociada con los arcos $(i, j) \in A$ y representa el *costo* (distancia, tiempo, etc) de ir desde el vértice $i$ hasta el vértice $j$. Generalmente se prohíben los arcos cíclicos $(i, i)$, y esto se logra definiendo $c_{ii} = +\infty$, $\forall\, i \in V$. Es fácil observar que si el grafo es no dirigido, la matriz de costos será simétrica $(c_{ij} = c_{ji},\ \forall\, i, j \in V)$. De lo contrario la matriz de costos será asimétrica.

En la mayoría de los casos prácticos, la matriz de costos satisface la propiedad de desigualdad triangular:

$$c_{ik} + c_{kj} \geq c_{ij}, \qquad \forall\, i, j, k \in V$$

Cada vértice $i$ $(i = 0, \dots, n)$ tiene asociada una demanda $d_i$ a ser entregada o recogida. Podemos imaginar la demanda como la cantidad de personas esperando en una parada

de autobús. El *depot* tiene una demanda ficticia $d_0 = 0$. Dado un subconjunto de vértices $U \subseteq V$, sea $d(U) = \sum_{i \in S} d_i$ la demanda total del subconjunto.

Se puede disponer de cualquier cantidad de vehículos iguales de capacidad $C$. Se asume que todos los vehículos están aparcados en el *depot*. Para asegurar viabilidad se asume que la demanda de cada vértice $d_i \leq C \; para \; todo \; i = 1, \dots, n$. Adicionalmente cada vehículo puede ejecutar a lo sumo una ruta.

### 2.2 REQUISITOS Y RESTRICCIONES

El DCVRP consiste en encontrar un conjunto $S$ de rutas que tenga costo mínimo. Cada ruta se corresponde con un único vehículo. El costo del conjunto $S$ se define como la suma de los costos de los arcos pertenecientes a las rutas. El conjunto debe cumplir las siguientes restricciones:

i. Cada ruta debe visitar el *depot* exactamente 2 veces, al principio y al final.
ii. Cada vértice (excepto el *depot*) debe ser visitado exactamente una vez por alguna ruta.
iii. Sea $V(r)$ el conjunto de vértices visitados por la ruta $r$ . $d\big(V(r)\big) \leq C \; para \; todo \; r \in S$. O en lenguaje natural, ninguna ruta de la solución puede exceder la capacidad de un vehículo.
iv. La suma de los costos de los arcos pertenecientes a una ruta $r$ tiene que ser menor que la restricción de distancia $D, para \; todo \; r \in S$.

## 3 ESTADO DEL ARTE

Como se expuso en la introducción, este problema se ha estado estudiando con intensidad durante más de medio siglo y ha sido tema de miles de publicaciones académicas. Por tanto, la siguiente recopilación constituye no más que una descripción general de los principales métodos identificados y para más detalles el lector tendrá que referirse a la literatura.

Es curioso notar que al igual que otros tantos problemas científicos famosos, las publicaciones alrededor del VRP no están homogéneamente distribuidas en el tiempo, sino que desde su aparición en 1959 ha habido períodos de efervescencia académica seguidos por etapas de silencio, que en algunos casos han llegado a durar cerca de una década. Entre un período fecundo y el siguiente, aparecen nuevos resultados teóricos y necesidades prácticas que son el detonante a una ola de investigaciones con enfoques y herramientas novedosas.

Cuando nos encontramos, como ahora, frente a un gran volumen de algoritmos y aproximaciones a un problema, surgen las siguientes preguntas. ¿Cómo categorizamos el conocimiento existente? ¿Cómo ordenamos el estudio de éste?

La respuesta inmediata a estas preguntas suele ser: cronológicamente. La aproximación cronológica tiene varias ventajas, siendo la más importante que la mayoría de las publicaciones utilizan resultados de publicaciones anteriores. Sin embargo luego de haber estudiado la muestra de algoritmos que vamos a ver, hemos considerado que para mayor claridad agruparemos los algoritmos por *tipo*.

Al igual que la mayoría de los problemas de optimización combinatorial, los algoritmos de solución al VRP pertenecen a alguno de los siguientes 3 tipos:

- Métodos de solución exacta.
- Heurísticas clásicas.
- Meta-heurísticas.

A qué tipo pertenece cada algoritmo, depende principalmente de la calidad del resultado esperado así como del tiempo de cálculo.

## 3.1 MÉTODOS DE SOLUCIÓN EXACTA

Los métodos de solución exacta son técnicamente los únicos que "resuelven" el problema en cuestión. El resto sólo ofrece soluciones aproximadas.

Profundicemos un poco más. Un algoritmo o método de solución al VRP puede ser calificado como "método de solución exacta" si y sólo si ofrece una solución *óptima* para cualquier configuración de la entrada. Lamentablemente la palabra "óptima" se ha devaluado en la mayoría de los entornos empresariales y se utiliza indiscriminadamente en lugar de "muy buena", o incluso en lugar de "mejor solución encontrada". Ambos usos del término son incorrectos.

Para poder afirmar que un algoritmo ofrece soluciones *óptimas*, es necesario haber demostrado matemáticamente que para cualquier configuración de la entrada *no existe* una solución mejor que la encontrada por el algoritmo en cuestión. Esta propiedad es evidentemente muy potente y difícil de cumplir. Generalmente va asociada a un gran costo espacial o temporal del algoritmo que en muchos casos requiere tiempos de cálculo inmanejables incluso para problemas pequeños.

Naturalmente la búsqueda de algoritmos de solución exacta fue la primera aproximación al VRP, y los algoritmos más antiguos fueron propuestos poco después de que el problema hubiera sido formalizado por primera vez. A continuación veremos algunos de los métodos de solución exacta más populares.

### *3.1.1 BÚSQUEDA EXHAUSTIVA*

Aunque la búsqueda exhaustiva (o *fuerza bruta*) no es precisamente popular, constituye la aproximación más intuitiva a este problema y es la base del potente branch-and-bound que veremos a continuación.

El nombre "*fuerza bruta"* es auto-descriptivo pero trataré de formalizar a qué nos referimos. Personalmente siempre me ha resultado más fácil comprender la búsqueda exhaustiva cuando pensamos en un árbol de *estados* en el que la raíz es el estado inicial y los hijos de cada nodo son los estados a los que se llega tomando la próxima *decisión*. A qué llamar una decisión es una parte importante del diseño de un algoritmo de fuerza bruta.

En este caso llamaremos "estado" a un grafo - no necesariamente conexo - con todos los nodos del problema, y llamaremos "decisión" a agregar una conexión entre nodos de dicho grafo. O sea, la raíz será un grafo vacío (i.e. sin arcos) y cada una de las maneras de agregar el próximo arco será un estado hijo. Durante la construcción o recorrido de este árbol de estados habrá que verificar que las decisiones que se toman no estén en contradicción con las restricciones del problema (detalladas en la sección 2.2). Naturalmente las soluciones serán las hojas, o sea, aquellos estados que no tienen hijos porque ya no quedan decisiones que tomar que no entren en contradicción con ninguna restricción. Para mejor compresión la Ilustración 3.1 contiene una representación parcial del árbol de estados de un problema pequeño.

No es difícil notar que el árbol de estados puede tener dimensiones inmanejables incluso para problemas pequeños. Como consecuencia, construirlo o recorrerlo es impracticable en la gran mayoría de los casos reales. Hay una gran variedad de optimizaciones que se pueden aplicar sobre una implementación de búsqueda exhaustiva, muchas de ellas simples de encontrar pero con mejoras despreciables en cuanto a rendimiento. Una de las mejores optimizaciones que se le puede aplicar a un algoritmo de búsqueda exhaustiva, consiste en el uso del método de branch-and-bound.

### *3.1.2 BRANCH AND BOUND*

El método de branch-and-bound ha sido extensamente utilizado en las últimas décadas para resolver las principales variantes del DCVRP. En muchos casos, esta técnica aun representa el estado del arte en cuanto al cálculo de soluciones exactas.

Como seguramente el lector ya se habrá percatado, el VRP es una extensión del famoso "Problema del viajero" (TSP por sus siglas en inglés) por lo que muchas aproximaciones para calcular soluciones exactas del VRP fueron heredadas del intenso y prolífero trabajo que se ha hecho entorno al TSP. En esto branch-and-bound no es una excepción, donde los algoritmos diseñados para ambos problemas tienen mucho en común. Específicamente, utilizan las mismas *relajaciones*.

Antes de entrar más en detalles debemos comentar en qué consiste la técnica de branch-and-bound, para luego ver cómo aplicarla a nuestro problema. En el campo de la optimización combinatorial casi todos los problemas pueden ser fácilmente modelados como un árbol de estados similar al visto anteriormente, donde los estados contienen un conjunto de variables con restricciones entre ellas y la relación padre hijo está dada por asignarle a alguna variable un valor de su dominio que no viole ninguna de las restricciones.

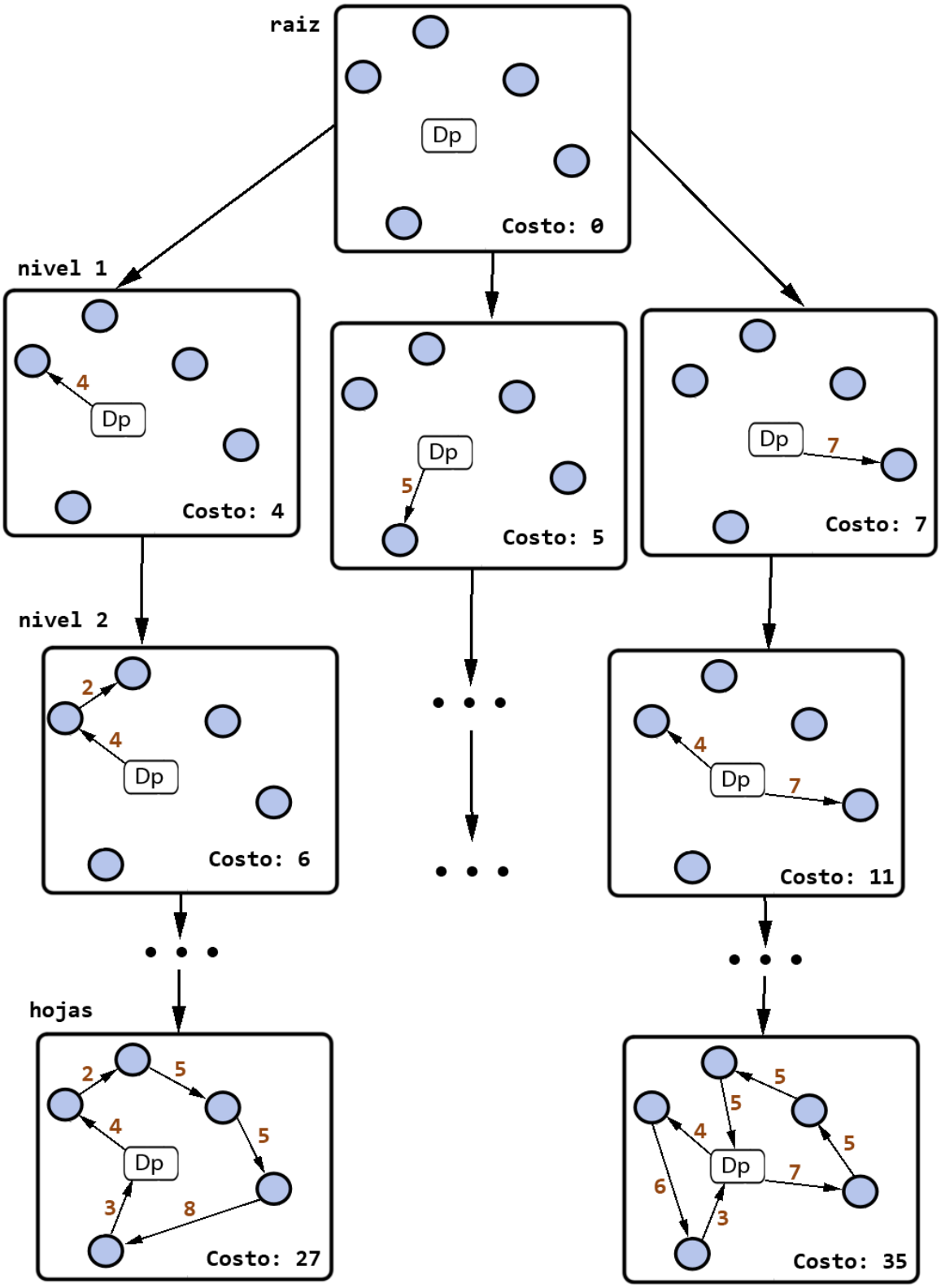


ILUSTRACIÓN 3.1 ÁRBOL DE ESTADOS.

La técnica de branch-and-bound consiste en encontrar cota mínima y máxima al costo de la solución posible luego de tomar cada decisión. Estaremos pensando en un árbol de estados, por lo que las palabras “nodo” y “estado” serán utilizadas indistintamente durante esta sección.

Olvidémonos un momento de cómo calcular estas cotas y vamos a concentrarnos en comprender su utilidad. Supongamos que por arte de magia conocemos para cada nodo $i$ de nuestro árbol de estados, su cota máxima $U_i$ y su cota mínima $L_i$. Esto quiere decir que toda solución (u hoja) que sea descendiente de un estado $i$ tendrá un costo de solución $C_s$ tal que $L_i \leq C_s \leq U_i$.

Adicionalmente debemos mantener actualizada una cota máxima global $U_g$ que será el mínimo de las cotas máximas de los nodos que hemos recorrido hasta ahora. Esta información es suficiente para discriminar subárboles enteros de estados que nos pueden consumir muchísimo tiempo de cálculo. Lo hacemos de la siguiente manera. Si para un nodo $i$ se cumple que $L_i > U_g$ podemos ignorar completamente a dicho nodo y a todos sus descendientes. Esto se debe a que existe otro estado $j$ (del cual se actualizó el valor de $U_g$) tal que todas las soluciones descendientes de $j$ tienen un costo de solución $C_s$ que cumple $C_s \leq U_g < L_i$. Por lo que todas las soluciones descendientes de $i$ serán peores que cualquier solución descendiente de $j$.

Lo explicado anteriormente no es más que la idea central de cualquier algoritmo de branch-and-bound. Una vez que se comprende, no es difícil notar que la efectividad de un algoritmo de este tipo depende principalmente de cuan ajustadas sean las cotas. Por ejemplo, se podría afirmar que $0$ es cota mínima de cualquier estado, pero esta cota es tan mala que será inútil para descartar cualquier trozo de la búsqueda.

Cómo encontrar cotas máximas suele ser simple (ej. implementación *greedy*), pero cómo encontrar cotas mínimas ajustadas es el verdadero arte de diseñar un algoritmo branch-and-bound y es lo que hace cada implementación dependiente del contexto. Aquí es donde entra el concepto de *relajación*.

Como regla general, los problemas de optimización consisten de variables con sus dominios, y restricciones entre ellas. *Relajar* un problema no es más que eliminar temporalmente una o más de las restricciones. Nótese que una solución del problema original cumple todas las restricciones del problema *relajado* ya que éstas son un subconjunto de las restricciones iniciales. Por lo tanto la solución óptima de un problema relajado nunca será peor que la solución óptima del problema original por lo que *relajar* un problema es ideal para encontrar cotas mínimas.

Claramente la técnica de *relajación* sólo es útil en aquellos casos en los que resolver el problema relajado sea trivial en comparación con el problema original. Si el problema relajado es igual de complejo que el problema original no hemos hecho nada. Por eso es tan delicado escoger las *relajaciones* adecuadas y decidir qué restricciones quitar.

Dos de las *relajaciones* más utilizadas para resolver las principales variantes del VRP son transformar al Problema de asignación (AP) o encontrar el Árbol recubridor mínimo (MST). Ambos problemas tienen algoritmos polinomiales conocidos.

### *3.1.3 PROGRAMACIÓN LINEAL*

La Programación lineal (LP) es un método matemático ampliamente utilizado para resolver problemas de optimización de casi cualquier naturaleza. Se trata de modelar un problema en forma de una función $f(x_1, x_2, \dots, x_n)$ que se quiere minimizar (o maximizar) sujeto a un conjunto de restricciones sobre las variables. Solo hay una condición. Tanto la función objetivo $f$ como las restricciones deben ser lineales. Las restricciones pueden ser igualdades o desigualdades, siempre que sean lineales.

La palabra *lineal* en matemáticas suele estar asociada con ecuaciones de primer grado, pero la definición formal es la siguiente. Una función $f$ es lineal, si y solo si cumple las siguientes dos propiedades:

i. Propiedad aditiva: $f(x + y) = f(x) + f(y)$
ii. Propiedad homogénea: $f(\alpha x) = \alpha f(x)$

La fortaleza de la Programación lineal radica en que una vez que se ha logrado modelar un problema de la manera descrita anteriormente, existe una gran cantidad de algoritmos y herramientas comerciales que resuelven el problema con un tiempo de cálculo polinomial en el tamaño de la entrada y en la cantidad de restricciones. Estamos hablando por ejemplo del algoritmo Simplex propuesto por George Dantzig en 1947, o del algoritmo del Punto Interior propuesto por Narendra Karmarkar en 1984. Esto hace que valga la pena echarle un vistazo a la Programación lineal como vía de solución al VRP, ya que las dos aproximaciones vistas anteriormente (Búsqueda exhaustiva y Branch and bound) tienen tiempos de cálculo exponenciales.

El reto entonces consiste en modelar el VRP en términos de Programación lineal. Veamos uno de los modelos más populares. Para ello vamos a necesitar una función $r(S)$ que se invoca sobre un conjunto de vértices $S$ y devuelve la mínima cantidad de vehículos de capacidad $C$ que son necesarios para cubrir la demanda de todos los vértices de $S$. Como los vértices no se pueden dividir este problema no es tan simple como suena. De hecho es un problema bien complejo conocido como *Bin Packing Problem* (BPP). No obstante es un problema bien estudiado y para lograr brevedad supongamos que contamos con una manera eficiente de resolver el BPP y la utilizaremos para implementar nuestra función $r(S)$.

Procedamos ahora a modelar el VRP en términos de Programación lineal. Recordemos que $V$ denota al conjunto de los vértices, $A$ al conjunto de los arcos, $c_{ij}$ a la matriz de costos y el *depot* está convenientemente asociado al vértice número 0. Adicionalmente necesitaremos una matriz booleana $x_{ij}$ que contendrá nuestras variables. La variable $x_{ij}$ tomará valor 1 si deseamos incluir el arco $(i, j) \in A$ en la solución y tomará valor 0 en caso contrario.

Dicho esto queremos minimizar:

$$f(x) = \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij}$$

Sujeto a las siguientes restricciones:

(i) $x_{ij} \in \{1, 0\}, \qquad \forall\, i, j \in V$

(ii) $\sum_{i \in V} x_{ij} = 1, \qquad \forall\, j \in V \setminus \{0\}$

(iii) $\sum_{j \in V} x_{ij} = 1, \qquad \forall i \in V \setminus \{0\}$

(iv) $\sum_{i \in V} x_{i0} = K$

(v) $\sum_{j \in V} x_{0j} = K$

(vi) $\sum_{i \notin S} \sum_{j \in S} x_{ij} \geq r(S), \qquad \forall\, S \subseteq V \setminus \{0\}, S \neq \emptyset$

La restricción (i) nos asegura que la matriz $x_{ij}$ sea booleana. Las restricciones (ii) y (iii) nos aseguran que cada nodo (excepto el depot) tenga exactamente un arco entrante y un arco saliente. Las restricciones (iv) y (v) nos aseguran que el depot tenga exactamente $K$ arcos entrantes y $K$ arcos salientes, donde $K$ es la cantidad de rutas del problema. Si no queremos prefijar la cantidad de rutas, estas dos restricciones se pueden ignorar. Las restricciones mencionadas hasta el momento nos aseguran que estamos buscando un grafo que conste solo de ciclos y que por cada nodo pueda pasar exactamente un ciclo, excepto por el depot que puede pasar cualquier cantidad de éstos. Estas restricciones no son suficientes ya que estaríamos permitiendo el grafo mostrado en la Ilustración 3.2:

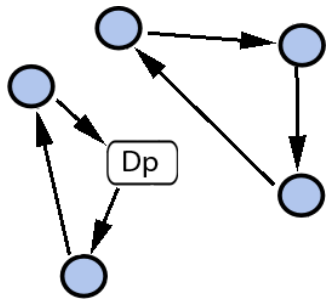


ILUSTRACIÓN 3.2 GRAFO INVÁLIDO QUE CUMPLE LAS PRIMERAS RESTRICCIONES

Es aquí cuando entra la restricción (vi). Esta restricción es particularmente ingeniosa y nos asegura a la misma vez que la solución sea conexa y que ninguna ruta exceda la capacidad de un vehículo. La restricción (vi) básicamente exige que para todo *corte* $(V \setminus S, S)$ la cantidad de arcos que cruzan el *corte* en una dirección sea mayor o igual que $r(S)$. Aunque no es nada evidente, esta restricción basta para asegurar las dos propiedades anteriores. Para ver la demostración el lector se puede referir a [1].

¿Entonces ya está? ¿Se puede resolver el VRP de manera eficiente mediante Programación lineal? La verdad es que hemos hecho una pequeña trampa. La restricción (vi) no es precisamente una restricción, sino que para cada $S \subseteq V$ genera una restricción nueva. Sabemos que $V$ tiene $2^{|V|}$ subconjuntos por lo que estamos hablando de una cantidad exponencial de restricciones. Como comentamos anteriormente Simplex y otros algoritmos de Programación lineal tienen un tiempo de cálculo polinomial en la cantidad de restricciones por lo que LP no es más que otro método de solución exponencial al VRP.

Además, en la práctica se ha comprobado que buenas implementaciones de branch-and-bound para el VRP son considerablemente más eficientes que aproximaciones de solución mediante Programación lineal.

## 3.2 HEURÍSTICAS CLÁSICAS

Una vez que los matemáticos comenzaron a descartar la posibilidad de encontrar algoritmos eficientes que encontraran soluciones *óptimas* al VRP, comenzó la investigación en el mundo de las *heurísticas.* Varias familias de heurísticas han sido propuestas para el VRP. En general pueden ser clasificadas en dos categorías: *heurísticas clásicas*, propuestas principalmente entre 1960 y 1990, y *meta-heurísticas*, cuyo crecimiento ha ocurrido en las últimas décadas.

Las *heurísticas clásicas* se caracterizan por explorar de forma limitada el espacio de búsqueda asegurando una calidad mínima en las soluciones y tiempos de cálculo muy cortos. Además, la mayoría de ellas son permisivas en cuanto a la inclusión de restricciones, lo que hace que actualmente éstas sean las más encontradas en paquetes comerciales. En las *meta-heurísticas*, por otra parte, el énfasis está en hacer una exploración profunda en las regiones más prometedoras del espacio de búsqueda. Estos métodos por lo general combinan técnicas complejas de búsqueda como definiciones de vecindad, estructuras de memoria o mezclado de soluciones vecinas. La calidad de las soluciones producidas por estos métodos es mucho mayor que la obtenida de las *heurísticas clásicas*, pero el precio a pagar es un tiempo de cálculo inflado. Además las *meta-heurísticas* suelen ser dependientes del contexto y suelen requerir un fino ajuste en los parámetros para cada situación diferente, lo que las hace de propósito específico. De alguna manera las *meta-heurísticas* pueden ser vistas como procedimientos de búsqueda sofisticados y no son más que mejoras naturales a las *heurísticas clásicas*. No obstante éstas hacen uso de varios conceptos novedosos no utilizados en las *heurísticas clásicas*, por lo que vamos a comentarlas en una sección aparte.

La mayoría de las heurísticas desarrolladas para el VRP se aplican directamente al CVRP y al DCVRP, incluso en aquellos casos en los que los autores del algoritmo no lo dijeron explícitamente. La mayoría también trabaja sin especificar de antemano la cantidad $K$ de rutas a crear, pero hay algunas excepciones. Esto se aclara en cada caso. La matriz de distancias que utilizan éstas heurísticas puede ser simétrica o asimétrica, pero se cuenta con muy pocos resultados experimentales para casos asimétricos.

La mayoría de las *heurísticas clásicas* utilizan alguna de las siguientes técnicas, o una mezcla de ellas:

1. Mezclar rutas existentes basándose en el concepto de *savings*. Esta palabra se ha utilizado para denotar cuanto mejora el costo global si se mezclan dos rutas.
2. Gradualmente insertar vértices en rutas existentes basándose en el *costo de inserción*.

### 3.2.1 *Algoritmo de Clarke and Wright*

El algoritmo de Clarke and Wright [2] es quizás la heurística más ampliamente conocida para el VRP. Se basa en el concepto de *savings*. Cuando dos rutas $(0, \dots, i, 0)$ y $(0, j, \dots, 0)$ pueden ser mezcladas en una y ruta $(0, \dots, i, j, \dots, 0)$, sin que esto viole las restricciones de capacidad o distancia, se genera un *saving* denotado por $s_{ij} = c_{i0} + c_{0j} - c_{ij}$. Esto se grafica con claridad en la Ilustración 3.3.

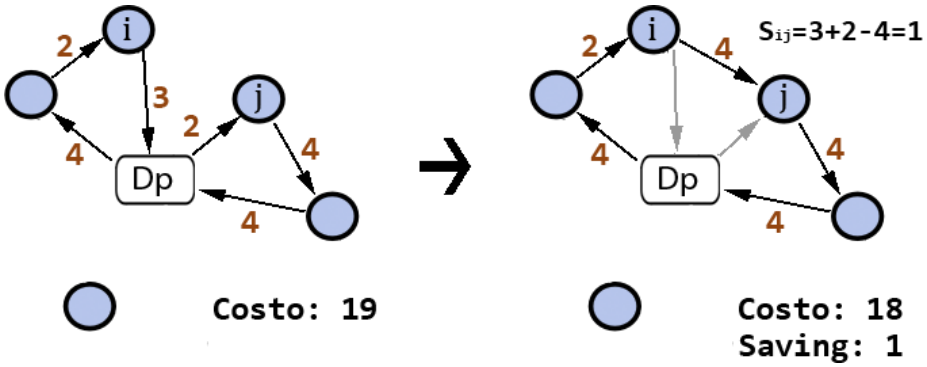


Ilustración 3.3 Representación gráfica del concepto de *saving*

Este algoritmo se aplica con naturalidad a la variante del VRP en la que la cantidad de rutas a crear es una variable, y funciona igual de bien con matrices de costo simétricas o asimétricas. Otra ventaja del algoritmo es que es relativamente simple de paralelizar. El flujo de ejecución es el siguiente:

**Paso 1** (calcular los savings). Se calculan los valores $s_{ij} = c_{i0} + c_{0j} - c_{ij},\ \forall\, i, j \in V \setminus \{0\}, i \neq j$. Se inicializan $n$ rutas de la forma $(0, i, 0),\ \forall\, i \in V$. Se ordenan los $n^2 - n$ *savings* calculados de mayor a menor (nótese que serían $\frac{n^2-n}{2}$ en el caso simétrico).

**Paso 2** (estrategia *greedy*). Se recorre la lista de *savings* de mayor a menor y se procede de la siguiente manera. Dado un *saving* $s_{ij}$, se determina si existen dos rutas tales que

una contenga el arco $(0, i)$ y otra contenga el arco $(j, 0)$, y que puedan ser mezcladas en una ruta sin violar las restricciones de capacidad o distancia. En caso positivo se mezclan dichas rutas mediante la eliminación de los arcos $(0, i)$ y $(j, 0)$, y la inserción del arco $(i, j)$.

Se encuentran grandes variaciones en la calidad de los resultados ofrecidos por esta heurística. En la Tabla 3.1 se muestran algunos resultados numéricos de Clarke and Wright. En la primera columna aparece el código que identifica a cada instancia estándar del VRP en la comunidad académica [3]. En la segunda y tercera columna aparecen el costo de la solución ofrecida por Clarke and Wright y el costo de la mejor solución conocida, respectivamente.

| ID de instancia | Clarke and Wright | Mejor solución conocida | Diferencia |
|---|---|---|---|
| E051-05e | 584,64 | 524,61 | 11,44 % |
| E076-10e | 900,26 | 835,26 | 7,78 % |
| E101-08e | 886,84 | 826,14 | 7,35 % |
| E101-10c | 833,51 | 819,56 | 1,70 % |
| E121-07c | 1071,07 | 1042,11 | 2,78 % |
| E151-12c | 1133,43 | 1028,42 | 10,21 % |
| E200-17c | 1395,74 | 1291,45 | 8,08 % |
| D051-06c | 618,40 | 555,43 | 11,34 % |
| D076-11c | 975,46 | 909,68 | 7,23 % |
| D101-09c | 973,94 | 865,94 | 12,47 % |
| D101-11c | 875,75 | 866,37 | 1,08 % |
| D121-11c | 1596,72 | 1541,14 | 3,61 % |
| D151-14c | 1287,64 | 1162,55 | 10,76 % |
| D200-18c | 1538,66 | 1395,85 | 10,23 % |

TABLA 3.1 RESULTADOS EXPERIMENTALES DE CLARKE AND WRIGHT

### *3.2.2 ALGORITMO DE BARRIDO*

El Algoritmo de barrido también se encuentra entre las *heurísticas clásicas* más populares. Ha clasificado en esta diminuta selección de algoritmos por su naturalidad y simpleza. Las primeras referencias conocidas a este algoritmo en la literatura están en un libro de Wren [4] y en un paper de Wren y Holliday [5]. No obstante el Algoritmo de barrido se atribuye con frecuencia a Gillett y Miller [6], quienes lo popularizaron. Hay dos características del Algoritmo de barrido que me gustaría mencionar antes de entrar en detalles.

La primera es que el algoritmo de barrido pertenece a una subfamilia de las *heurísticas clásicas* conocida como "Métodos de dos fases". Los métodos de dos fases se caracterizan por agrupar primero y enrutar después. O sea, en la primera fase forman grupos de vértices que serán recorridos dentro de una misma ruta, y en la segunda fase resuelven un problema del viajero (TSP) para cada grupo. Esta técnica es conceptualmente diferente de los algoritmos que construyen la solución de forma incremental, como lo hace el Algoritmo de Clarke and Wright. La fortaleza de los algoritmos de dos fases radica en la reutilización del extenso trabajo que se ha hecho alrededor del TSP. El problema se reduce entonces a agrupar los vértices de la mejor manera.

La segunda característica es más bien una restricción. El Algoritmo de barrido solo es aplicable a versiones del problema conocidas como *Euclidean* VRP, o *Planar* VRP. Se trata de los problemas en los que los vértices pueden ser transportados de alguna manera a un plano de coordenadas cartesianas y el costo entre ellos está dado por la llamada *distancia euclidiana* ($\sqrt{\Delta x^2 + \Delta y^2}$).

El algoritmo consiste en formar grupos de vértices que quepan en un vehículo, rotando una línea centrada en el *depot*. Una ruta luego es construida para cada grupo mediante la solución de un TSP. Una implementación simple de este método es la siguiente. Asumamos que cada vértice $i$ está representado por sus coordenadas polares $(\theta_i, \rho_i)$, donde $\theta_i$ es el ángulo con respecto al eje $x$ positivo y $\rho_i$ es la distancia al origen. Ordenamos todos los vértices (menos el *depot*) de mayor a menor valor de $\theta_i$.

**Paso 1** (inicialización): Creamos un grupo vacío.

**Paso 2** (construcción de grupo): Comenzando por el vértice sin grupo asignado de menor $\theta_i$, vamos asignando los vértices al nuevo grupo, mientras la demanda total no exceda la capacidad máxima de un vehículo. Si aún quedaran vértices sin grupo, regresar al Paso 1.

**Paso 3** (optimización de rutas): Para cada uno de los grupos creados se calcula de manera independiente la mejor manera de recorrerlos. Esto se hace mediante cualquier técnica de solución al TSP, ya sea algoritmo solución exacta o heurística. Este paso se puede paralelizar con facilidad.

La Ilustración 3.4 muestra una representación gráfica de lo explicado anteriormente. En la Fase 1 se van agrupando los vértices en orden del ángulo $\theta_i$. Cada color representa un grupo diferente. En la Fase 2 se calcula la solución al problema del viajero (TSP) para cada grupo.

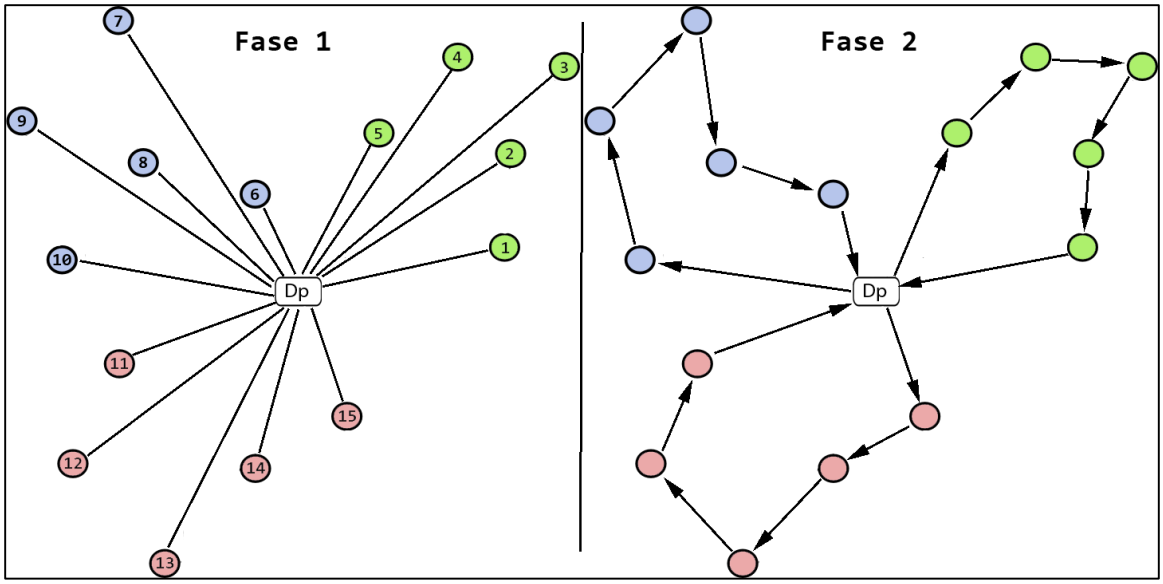


ILUSTRACIÓN 3.4 REPRESENTACIÓN GRÁFICA DEL ALGORITMO DE BARRIDO

## 3.3 META-HEURÍSTICAS

Varias *meta-heurísticas* han sido propuestas para el VRP desde 1990. Se tratan de métodos de solución que exploran el espacio de soluciones y con frecuencia incluyen algunas de las técnicas de construcción u optimización de rutas utilizadas en *heurísticas clásicas*. La gran diferencia con las aproximaciones clásicas es que las *meta-heurísticas* permiten soluciones inválidas a lo largo del curso de un proceso de búsqueda. Esto ha demostrado ser de gran utilidad para analizar regiones del espacio de soluciones que las aproximaciones clásicas ignoraban. Las mejores *meta-heurísticas* conocidas para el VRP suelen encontrar mejores soluciones que las aproximaciones clásicas, pero también tienden a consumir más tiempo de cálculo.

En la literatura hemos encontrado principalmente seis tipos de *meta-heurísticas* que han sido aplicadas al VRP:

1. Simulated Annealing
2. Deterministic Annealing
3. Tabu Search
4. Algoritmos genéticos
5. Algoritmos de hormigas
6. Redes de neuronas

Los primeros tres tipos de algoritmos se caracterizan por comenzar desde una solución inicial $x_1$ y moverse en cada iteración $t$ desde la solución $x_t$ hacia la solución $x_{t+1}$ que pertenece a la vecindad de $x_t$ denotada por $N(x_t)$. Adicionalmente, si $f(x)$ denota el costo de la solución $x$, estos algoritmos no aseguran que $f(x_t) \geq f(x_{t+1})$. Como resultado se debe tener especial cuidado para evitar quedar atrapado en un ciclo de soluciones.

En el caso de algoritmos genéticos, un grupo de soluciones se analiza en cada iteración. Cada grupo se deriva del anterior, mezclando las soluciones más prometedoras y descartando las de peor costo.

Algoritmos de hormigas es una aproximación de construcción incremental en la que varias soluciones se crean en cada iteración utilizando parte de la información recopilada en iteraciones anteriores.

Redes de neuronas se basa en simular mecanismos de aprendizaje que de manera gradual ajustan los costos de tomar cada decisión hasta que se encuentra alguna solución aceptable. Las reglas que rigen cada búsqueda varían en cada caso y deben ser personalizadas a la medida de cada problema. Adicionalmente, una buena cantidad de creatividad y experimentación son necesarias para utilizar éstos algoritmos con éxito.

El propósito de esta sección es exponer con un poco más de detalles al menos dos de las aproximaciones mencionadas que más se utilicen.

### *3.3.1 SIMULATED ANNEALING*

En la iteración $t$ de *simulated annealing*, se escoge de manera aleatoria una solución $x$ que pertenezca a la vecindad de la solución actual $N(x_t)$. Si $f(x) \leq f(x_t)$, entonces se toma la próxima solución $x_{t+1} = x$, de lo contrario,

$$x_{t+1} = \begin{cases} x & con\ probabilidad \quad p_t \\ x_t & con\ probabilidad\ 1 - p_t \end{cases}$$

Donde $p_t$ usualmente es una función inversamente proporcional a número de iteración $t$ y la diferencia de costo entre soluciones $f(x) - f(x_t)$. Es común ver que $p_t$ se defina como,

$$p_t = \exp(-\frac{[f(x) - f(x_t)]}{\theta_t})$$

Donde $\theta_t$ denota la *temperatura* de la iteración $t$. Es de éste término que sale el nombre *simulated annealing.* La regla utilizada para definir el valor de $\theta_t$ recibe el nombre de *planificación de enfriamiento.* Por lo general la temperatura $\theta_t$ es inversamente proporcional al número de iteración $t$. Inicialmente a $\theta_t$ se le asigna un valor $\theta_1 > 0$ y cada $T$ iteraciones el valor se multiplica por un factor $\alpha$ tal que $0 < \alpha < 1$, de manera tal que la probabilidad de aceptar una solución de peor costo vaya disminuyendo con el tiempo. Tres condiciones de parada utilizadas con frecuencia son:

1. El costo de la mejor solución encontrada no ha mejorado más de $\pi_1\%$ durante los últimos $k_1$ ciclos enteros de $T$ iteraciones.
2. La cantidad de soluciones aceptadas ha sido menos que el $\pi_2\%$ de $T$ durante los últimos $k_2$ ciclos enteros de $T$ iteraciones.
3. Se han ejecutado en total $k_3$ ciclos enteros de $T$ iteraciones.

Una de las primeras implementaciones de *simulated annaeling* (SA) fue propuesta por Osman [7] en 1993. La Tabla 3.2 muestra una comparación de los resultados ofrecidos por dicha implementación. Los problemas mostrados en la comparación son los mismos que los que hemos utilizado anteriormente para comparar el Algoritmo de Clarke and Wright.

| ID de instancia | Implementación de SA por Osman | Mejor solución conocida | Diferencia |
|---|---|---|---|
| E051-05e | 528 | 524,61 | 0,65 % |
| E076-10e | 838,62 | 835,26 | 0,40 % |
| E101-08e | 829,28 | 826,14 | 0,38 % |
| E101-10c | 826 | 819,56 | 0,79 % |
| E121-07c | 1176 | 1042,11 | 12,85 % |
| E151-12c | 1058 | 1028,42 | 2,88 % |
| E200-17c | 1378 | 1291,45 | 6,70 % |
| D051-06c | **555,43** | 555,43 | 0,00 % |
| D076-11c | **909,68** | 909,68 | 0,00 % |
| D101-09c | 866,75 | 865,94 | 0,09 % |
| D101-11c | 890 | 866,37 | 2,73 % |
| D121-11c | 1545,98 | 1541,14 | 0,31 % |
| D151-14c | 1164,12 | 1162,55 | 0,14 % |
| D200-18c | 1417,85 | 1395,85 | 1,58 % |

TABLA 3.2 RESULTADOS EXPERIMENTALES DE *SIMULATED ANNAELING*

Generalmente las implementaciones de *simulated annealing* producen buenos resultados. No obstante, en algunos casos producen resultados bastante malos y en otros simplemente calculan la mejor solución conocida. En la tabla vemos como la implementación de Osman produjo un resultado malo para el problema E121-07c, y produjo la mejor solución conocida en los problemas D051-06c y D076-11c.

*Deterministic annealing* funciona de manera similar. La única diferencia con *smiulated annealing* es que se obvia el uso de probabilidades y en su lugar se utiliza una regla determinista para aceptar o no una próxima solución.

### *3.3.2 TABU SEARCH*

En *Tabu search*, secuencias de soluciones son analizadas al igual que en *simulated annealing*, pero la siguiente solución $x_{t+1}$ será siempre el mejor vecino de la solución actual $x_t$. Para evitar ciclos, las soluciones que fueron analizadas recientemente se marcan como prohibidas, o *tabú*, durante un número de iteraciones. Para disminuir los

requerimientos de tiempo y memoria, es necesario almacenar ciertos atributos de las soluciones *tabú* en lugar de las soluciones en sí.

Una de las mejores implementaciones conocidas que utilice la técnica de *Tabu search*, es el algoritmo *Taburoute*, propuesto por Gendreau, Hertz y Laporte [8] en 1994. Con respecto a otras implementaciones, ésta tiene algunas característica innovadoras. Una estructura de vecindad se define como todas las soluciones que se pueden obtener a partir de la solución actual, eliminando algún vértice de su ruta actual e insertándolo en otra ruta que contenga al menos uno de sus $p$ vecinos más cercanos. La eliminación y reinserción de vértices se hace siguiendo la estrategia *GENI*, propuesta por los mismos autores dos años antes [9] para el Problema del viajero (TSP).

Durante el proceso de reubicación de vértices se pueden eliminar rutas existentes o crear rutas nuevas. Otra característica importante de *Taburoute* es que el proceso de búsqueda analiza soluciones que pueden ser inválidas con respecto a las restricciones de capacidad o distancia máxima de ruta. Específicamente la función objetivo contiene dos términos de penalización, uno que mide el exceso total de capacidad y otro que mide el exceso total de distancia. Estos dos términos contienen factores de penalización que se auto ajustan durante la ejecución. Cada 10 iteraciones, los factores se dividen a la mitad si las últimas 10 soluciones fueron todas válidas o se duplican si fueron todas inválidas. La idea es que los factores de penalización no sean muy débiles ni muy restrictivos. De esta manera se asegura que cada 10 iteraciones se analizan soluciones válidas e inválidas, disminuyendo la probabilidad de quedar atrapado en un mínimo local. En algunos puntos del proceso de búsqueda, *Taburoute* optimiza todas las rutas de la solución utilizando el método *Unstringing and Stringing* (US) propuesto por estos autores en el paper mencionado anteriormente [9].

A diferencia de otros algoritmos, *Taburoute* no utiliza una lista de soluciones *tabú* sino que en su lugar utiliza un sistema de etiquetas para evitar ciclos. Cuando un vértice es eliminado de una ruta $r$ en la iteración $t$, su reinserción en la ruta $r$ queda prohibida hasta la iteración $t + \theta$, donde $\theta$ es un entero aleatorio en el intervalo $[5,10]$. Otra característica de este algoritmo es la llamada estrategia de *diversificación.* Consiste en penalizar aquellos vértices que se mueven con frecuencia para darle oportunidad a los vértices de menor movilidad.

La Tabla 3.3 muestra los resultados experimentales de *Taburoute* sobre las mismas instancias de VRP que hemos estado utilizando hasta ahora. Como se aprecia, los resultados de *Taburoute* son muy buenos, llegando a encontrar la mejor solución conocida para más de la mitad de las instancias (resultados en **negrita**). Adelantamos que este algoritmo será incluido en el proyecto, por lo que más adelante profundizaremos en detalles de implementación.

| ID de instancia | Taburoute | Mejor solución conocida | Diferencia |
|---|---|---|---|
| E051-05e | **524,61** | 524,61 | 0,00 % |
| E076-10e | 835,32 | 835,26 | 0,01 % |
| E101-08e | **826,14** | 826,14 | 0,00 % |
| E101-10c | **819,56** | 819,56 | 0,00 % |
| E121-07c | **1042,11** | 1042,11 | 0,00 % |
| E151-12c | 1031,07 | 1028,42 | 0,26 % |
| E200-17c | 1311,45 | 1291,45 | 1,55 % |
| D051-06c | **555,43** | 555,43 | 0,00 % |
| D076-11c | **909,68** | 909,68 | 0,00 % |
| D101-09c | **865,94** | 865,94 | 0,00 % |
| D101-11c | **866,37** | 866,37 | 0,00 % |
| D121-11c | 1545,93 | 1541,14 | 0,31 % |
| D151-14c | 1162,89 | 1162,55 | 0,03 % |
| D200-18c | 1404,75 | 1395,85 | 0,64 % |

TABLA 3.3 RESULTADOS EXPERIMENTALES DE *TABUROUTE*

# 4 DISEÑO Y DESARROLLO DE LA SOLUCIÓN

En este punto hemos completado el primer paso. Hemos hecho un recorrido por la literatura publicada alrededor del problema y estamos listos para diseñar una solución que utilice una o varias de las técnicas estudiadas. Es común ver como en entornos empresariales se reinventa la rueda una y otra vez, simplemente por la falta de hábito de consultar la literatura de vez en cuando. Entiendo que un equilibrio debe ser encontrado entre la teoría y la práctica, y que las empresas son en definitiva negocios en los que la rentabilidad es un requisito. Pero en problemas tan complejos como el VRP, investigar o no la literatura puede ser la diferencia entre la frustración y hacer un producto funcional.

Teniendo una visión general de cómo lidiar con ésta familia de problemas, veamos los requisitos del problema que tenemos entre manos.

## 4.1 REQUISITOS DEL PROBLEMA A RESOLVER

A la fecha de mi incorporación de Goal Systems, varios clientes estaban solicitando soluciones a problemas muy similares. El problema variaba de un cliente a otro, pero todos tenían en común el deseo de optimizar los recursos necesarios para ejecutar una recogida de personal. Diferentes maquillajes del problema podían ser la recogida de

estudiantes que van al colegio, u obreros que van a una fábrica; diferencias que lógicamente hablando no modifican el problema.

Parte de la labor de Goal Systems es comprender y formalizar los requisitos del cliente, ya que muchas veces ni éste los tiene claro. Aunque las restricciones del problema no era definitivas, y los clientes pueden (y suelen) cambiar las especificaciones varias veces durante el desarrollo de un producto, tuvimos que hacernos de una hoja de especificaciones formales que tomaríamos como guía inicial por la cual regir nuestro desarrollo. Veamos dichas especificaciones.

Estamos en un escenario en el que se quiere recoger a un gran número de personas distribuidas por la ciudad y se quieren dejar en un "destino" antes de una hora límite. Existen además varios puntos de recogida en la ciudad (en adelante "paradas") en las que una cantidad específica de personas va a estar esperando cada día para ser transportadas al destino. Adicionalmente cada parada tiene asignado un tiempo de espera obligatorio. O sea, si un autobús se detiene a recoger a las personas de una parada, deberá ser considerada una demora de varios minutos. Dicha demora se especifica en cada caso.

Se dispone de una cantidad "ilimitada" de conductores y autobuses con una capacidad fija para recoger al personal. Los conductores pueden comenzar su recorrido en cualquier parada y deben terminar en el destino pasando por tantas paradas como sea necesario.

Existe una hora mínima antes de la cual no se puede efectuar ninguna recogida. Existe una hora límite antes de la cual todas las personas tienen que estar en el destino. Adicionalmente existe una distancia máxima que puede recorrer un mismo vehículo.

Como de costumbre, el recurso más caro es el personal. Por tanto, queremos minimizar la cantidad de horas consumidas por los conductores. O sea, queremos minimizar la sumatoria de los tiempos de recorrido de cada ruta de la solución. Sujeto a esto queremos minimizar la cantidad total de rutas necesarias.

Veamos cual debería ser la entrada y la salida de nuestra aplicación.

**Entrada**

- Capacidad de los autobuses.
- Hora de inicio de recogida.
- Hora de fin de recogida.
- Distancia máxima de ruta.
- Colección de paradas. Para cada parada se especifica:
    - Cantidad de personas a ser recogidas.
    - Tiempo de estancia obligatorio.
- Colección de enlaces entre paradas. Para cada enlace se especifica:
    - Parada de origen.

- Parada de destino.
- Tiempo de recorrido.
- Distancia de recorrido.

**Salida**

- Tiempo total utilizado.
- Distancia total recorrida
- Colección de rutas a recorrer. Para cada ruta se especifica:
  - Paradas que recorre.
  - Cuantas personas recoge en cada parada

Por lo visto estamos lidiando con una versión del VRP que específica distancia y tiempo máximo de ruta, además de capacidad de carga de los vehículos. Específicamente se trata de un DCVRP. Hay que mencionar que nuestro problema tiene varias diferencias con el VRP clásico en las que abundaremos más adelante. No obstante el DCVRP será el problema que tendremos en mente a la hora de diseñar y escoger nuestros algoritmos. Ya hemos visto varios algoritmos y heurísticas para resolver este problema con diferentes niveles de complejidad y calidad de resultados. Tenemos el desafío de organizar un desarrollo incremental; obtener resultados en poco tiempo para solicitar realimentación a los clientes y la misma vez dejar el terreno preparado para incorporar técnicas más complejas.

Primero veamos de qué tecnologías y entornos de desarrollo disponemos. Además comentaremos cual será de manera general la arquitectura de la aplicación y cómo será el flujo de información.

## 4.2 ASPECTOS TÉCNICOS

A diferencia de otros tipos de software, los productos comerciales de optimización suelen consumir tiempos de cálculo que rondan las horas o incluso días. Esto hace que el rendimiento sea siempre el cuello de botella, por lo que la mayoría de las decisiones de tecnologías y lenguajes a utilizar responden a esta necesidad.

Hasta la fecha de hoy (septiembre del 2013) en Goal Systems se utiliza C++98 como lenguaje principal, ya que este ofrece un buen equilibrio entre rendimiento y legibilidad. Como entorno de desarrollo se utiliza Visual Studio 2005 Professional Edition, por lo que obviamente el compilador de C++ utilizado es VC++. Visual Studio tiene una gran cantidad de defensores y detractores. La realidad es que es el IDE insignia en algunas compañías grandes que utilizan C++ a diario como Blizzard. Goal Systems también lo ha adoptado como la mejor opción para hacer grandes desarrollos en C++. Como consecuencia, en aquellos desarrollos donde el rendimiento no es crítico y por lo tanto C++ no es utilizado, se utilizan otras tecnologías de Microsoft como Windows Form, Windows Presentation Fundation, ASP.NET y el difunto Silverlight.

Antes de continuar, quisiera comentar que los acuerdos de confidencialidad que firmé con Goal Systems no me permiten publicar el proyecto en su totalidad. No obstante, tengo la posibilidad de anexar a este documento porciones del código para su mejor entendimiento.

Este proyecto comenzó sobre Visual Studio 2005 y fue posteriormente portado a Visual Studio 2010 Ultimate Edition. El objetivo era hacer un núcleo de cálculo en C++ que estuviese completamente aislado de fuentes de datos y de interfaces gráficas, que fuese lo más versátil posible para que se le pudiera enchufar cualquier entrada y salida.

De cualquier manera, algún banco de pruebas era necesario para realizar el desarrollo de manera eficiente. Debíamos fabricar algún tipo de interfaz que permitiera definir un DCVRP con facilidad y adicionalmente transfiriera el problema definido al núcleo de cálculo para luego graficar los resultado obtenidos. C++ no es particularmente famoso por su facilidad para desarrollar interfaces gráficas simples, por lo que decidimos utilizar Windows Form. Tres días y 700 líneas de C# más tarde teníamos algo que lucía así. (Ilustración 4.1)

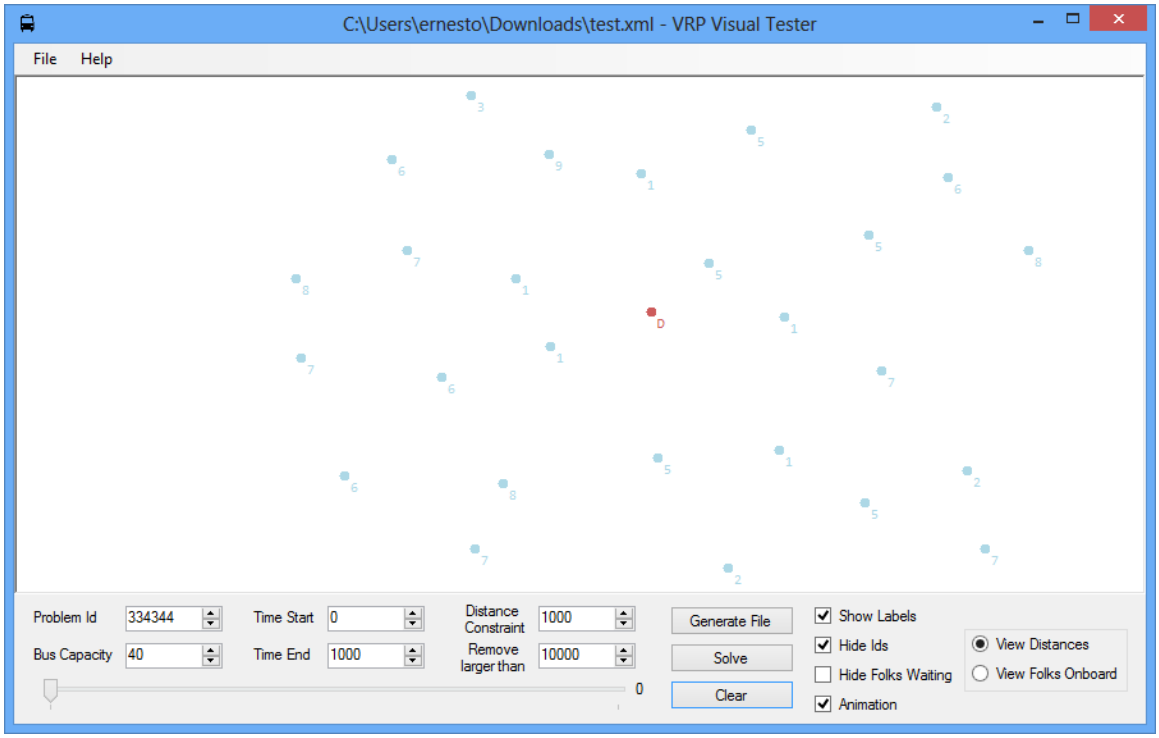


ILUSTRACIÓN 4.1 INTERFAZ GRÁFICA DE PRUEBAS

Se trata de una aplicación sencilla que permite definir a punta de clic todos los parámetros del problema. Los puntos verde claro son las paradas y el punto rojo es el *depot.* La cantidad de personas esperando en cada parada está representada con un número pequeño junto a estas. Se pueden agregar o eliminar paradas con el mouse además de modificar las propiedades de las paradas existentes. Adicionalmente la aplicación permite salvar y cargar instancias del problema para poder comprobar como

cambios algorítmicos afectan el resultado para una instancia en específico. El código principal de ésta aplicación está en el anexo MainForm.cs.

En Visual Studio un "proyecto" es una parte modular de un producto que genera uno o más binarios. Una "solución" puede contener uno o más "proyectos", incluso en lenguajes diferentes, que se comunican entre ellos utilizando los mecanismos ofrecidos para ello. Según lo que hemos comentado hasta ahora, nuestra solución tendría al menos un proyecto en .NET y tantos proyectos de C++ como fuesen necesarios para lograr un producto modular y organizado. La Ilustración 4.2 muestra cómo quedaría conformada la solución.

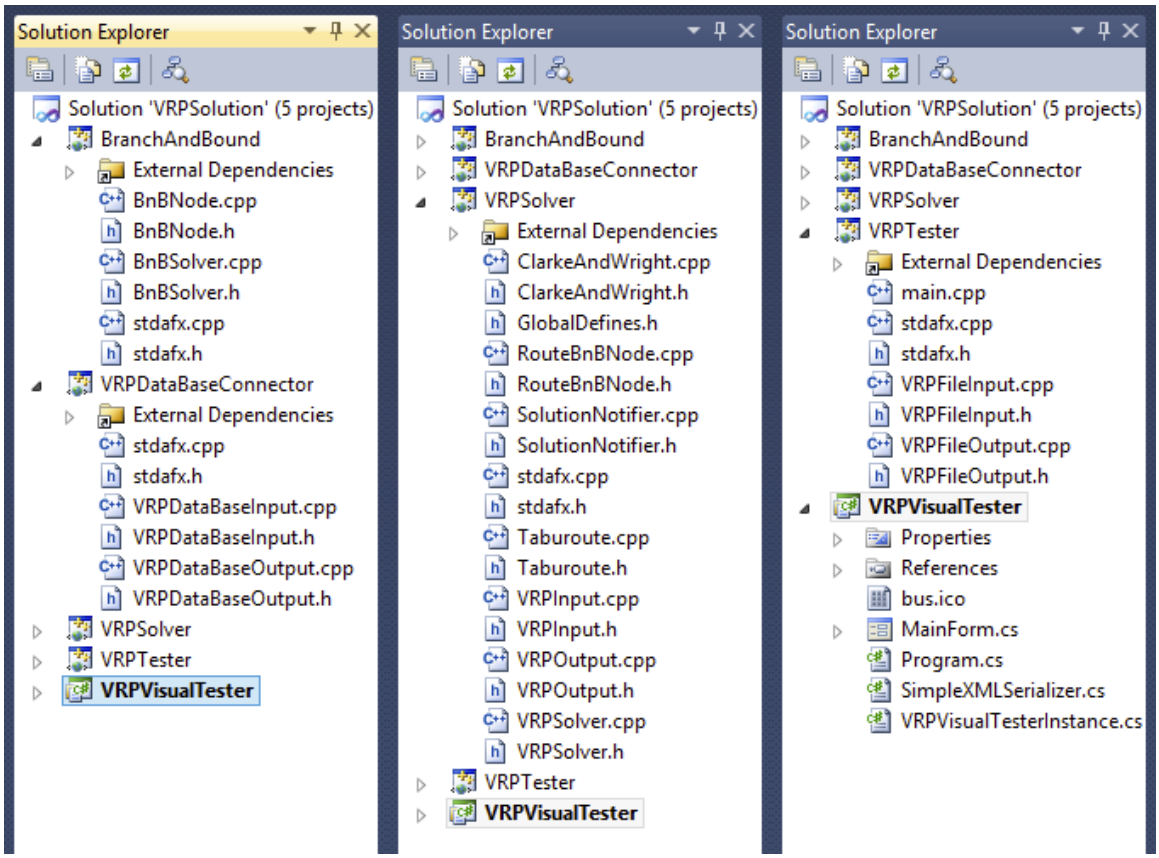


ILUSTRACIÓN 4.2 ORGANIZACIÓN EN PROYECTOS DE LA SOLUCIÓN

En total serían 5 proyectos. VRPVisualTester es la aplicación de Windows Form que vimos anteriormente. Los otros 4 proyectos serían hechos en C++. Los mencionamos a continuación en orden de importancia. VRPSolver es el proyecto más importante. Contiene parte de los algoritmos vistos anteriormente. Además funciona con clases abstractas VRPInput y VRPOutput, lo que hace que sea fácilmente conectable con diferentes fuentes de datos o interfaces gráficas. VRPTester es una interfaz de consola simple que se conecta con VRPSolver para hacer pruebas. Contiene un par de clases que permiten conectar la entrada y la salida de VRPSolver a ficheros XML. VRPDataBaseConnector contiene las clases y la configuración necesaria para conectar la fuente de datos de VRPSolver a una base de datos Oracle. El proyecto BranchAndBound

es una implementación genérica de la estrategia que le da nombre. Se decidió extraer del proyecto principal porque al ver los resultados que ofrecía, otros productos de la empresa quisieron incorporarlo. Por último, si el lector está familiarizado con Visual Studio sabrá que los ficheros stdafx.cpp y stdafx.h son los llamados *precompiled headers* que reducen considerablemente el tiempo de compilación.

Nótese que los proyectos hechos en C++ no utilizan "C++ manejado" ya que esto tendría implicaciones negativas de rendimiento. Como consecuencia las opciones de comunicación entre los procesos de C++ y de .NET son escasas e incómodas de utilizar. En este caso hemos decidido conectarlos utilizando la interfaz de consola VRPTester como intermediaria.

La aplicación VRPVisualTester ya utilizaba clases de serialización de objetos a XML para la gestión de salvar y cargar instancias. Se han reutilizado dichas clases para serializar una instancia del problema a un fichero XML del que el proceso VRPTester pudiera extraer los datos. Una instancia del VRP quedaría serializada de una manera similar a la siguiente:

```
1.  <?xml version="1.0"?>
2.  <VRPVisualTesterInstance xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3.    <ProblemId>334344</ProblemId>
4.    <BusCapacity>40</BusCapacity>
5.    <TimeStart>0</TimeStart>
6.    <TimeEnd>1000</TimeEnd>
7.    <DistanceConstraint>1000</DistanceConstraint>
8.    <Stops>
9.      <VRPVisualStop>
10.       ...
```

El proceso VRPTester sería responsable de parsear esos ficheros y crear objetos que sirviesen de entrada a los algoritmos de VRPSolver. De forma análoga el flujo de datos correspondientes a la salida del algoritmo iría en dirección contraria al de entrada. El proceso VRPTester tendría que serializar a XML las instancias de soluciones lógicas producidas por VRPSolver. Luego haciendo uso de la clase FileSystemWatcher de la plataforma .NET, el proceso VRPVisualTester sería notificado cada vez que una nueva solución hubiese sido producida y procedería a cargar esta en memoria para representarla gráficamente.

Ya que hemos visto como fue organizada la solución de forma general, pasemos a ver cuál fue la estrategia seguida para el desarrollo algorítmico del módulo VRPSolver, así como los resultados que se fueron obteniendo.

## 4.3 SOLUCIÓN INICIAL MEDIANTE CLARKE AND WRIGHT

Es una característica bastante común a la mayoría de las empresas de desarrollo de software que no es aceptable trabajar durante semanas en un proyecto sin resultados visibles, resultados que se puedan probar y se puedan mostrar a los clientes, para tener una noción de en qué dirección se está avanzando y tener realimentación de los clientes. Quizás sea esa la razón por la que las metodologías ágiles han tenido tanta popularidad en las últimas décadas.

Este proyecto no era una excepción. Yo estaba trabajando completamente aislado y me hubiera costado mucho justificar semanas de trabajo sin resultados bajo el argumento de estar trabajando en algo sofisticado. Por eso tenía el propósito de ofrecer resultados lo antes posible y trabajar de manera incremental a partir de estos.

El Algoritmo de Clarke and Wright fue uno de los primeros que descubrí cuando comenzaba mi corto viaje por la literatura del VRP. Éste cumplía todos los requisitos que estaba buscando para empezar. Ofrecía resultados decentes (Tabla 3.1), era simple de implementar y muy fácil de comprender en caso de que tuviera que hacer una pequeña presentación interna explicando cómo se calculaban estos primeros resultados. Así fue que este algoritmo fue asumido como primer paso.

No obstante, no era tan simple como implementar este algoritmo tal cual aparece en la literatura. Nuestro problema tiene varias diferencias importantes con respecto a la versión clásica del VRP, siendo la más importante que las rutas son "abiertas". Recordemos que entre los requisitos y restricciones del VRP clásico, está que las rutas deben comenzar y terminar en el *depot*, sin embargo nuestro problema especifica claramente que los conductores pueden comenzar su recorrido en la parada que se desee y terminar en el *depot* pasando por tantas paradas como sea necesario. La Ilustración 4.3 muestra cómo quedaría la solución de ejemplo que hemos visto al principio de este documento si las rutas fuesen abiertas.

Como vimos anteriormente el Algoritmo de Clarke and Wright trabaja desde el principio con rutas cerradas y esta propiedad es invariante durante toda la ejecución. Para adaptarlo a nuestro problema tendríamos que redefinir el concepto de *saving* de la siguiente manera. Cuando dos rutas $(\dots, i, 0)$ y $(j, \dots, 0)$ pueden ser mezcladas en una y ruta $(\dots, i, j, \dots, 0)$, sin que esto viole las restricciones de capacidad o distancia, se genera un *saving* denotado por $s_{ij} = c_{i0} - c_{ij}$. La Ilustración 4.4 muestra el concepto de *saving* adaptado.

Sin embargo, durante la implementación no se modificó el cálculo de *savings*, sino que se insertaron nodos virtuales tal que para cada nodo virtual $v$ se cumplía:

$$c_{0v} = 0 \; y \; c_{vi} = 0, \qquad \forall i \in V \setminus \{0\}$$

Como los costos del problema son positivos, esta propiedad basta para asegurar que cada ruta escoja un nodo virtual como primero a visitar. Luego simplemente hay que

eliminar los nodos virtuales de la solución final. Estos detalles de implementación se pueden observar en el anexo ClarkeAndWright.cpp.

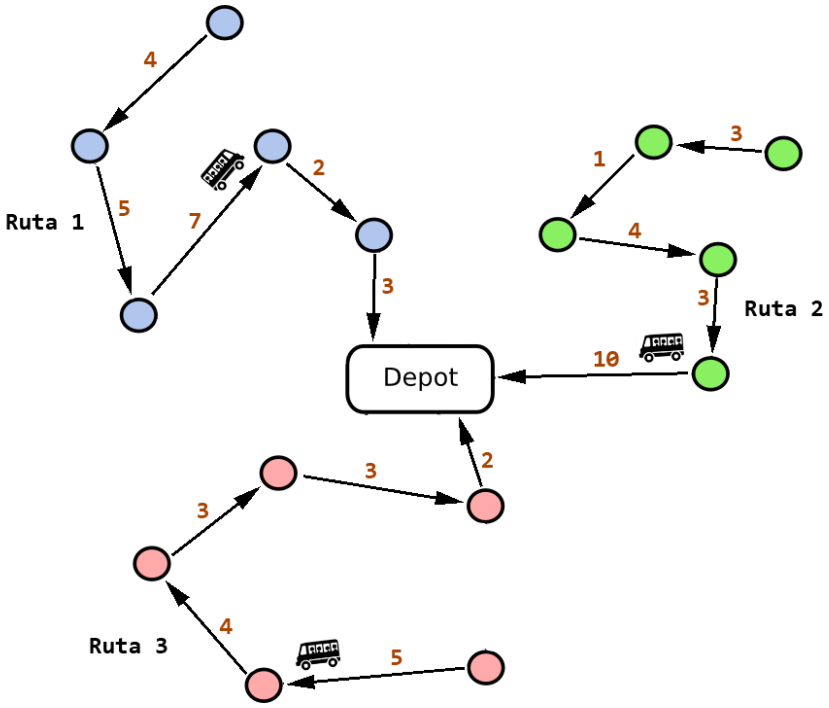


ILUSTRACIÓN 4.3 ADAPTACIÓN DEL EJEMPLO INTRODUCTORIO

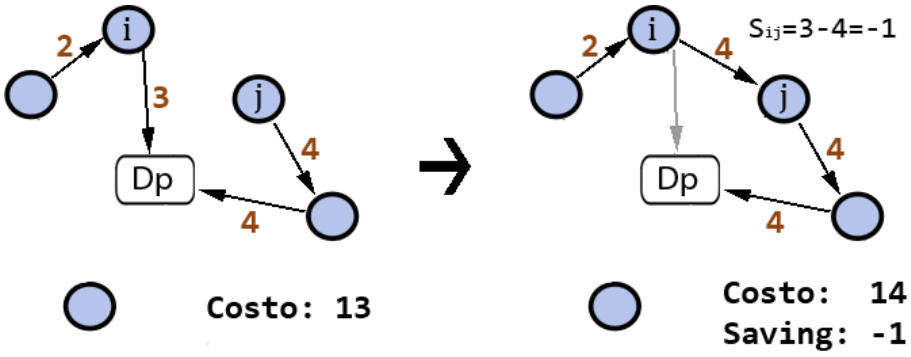


ILUSTRACIÓN 4.4 ADAPTACIÓN DEL CONCEPTO DE *SAVING*

La Ilustración 4.5 muestra los resultados ofrecidos por la aplicación tras la esta implementación de Clarke and Wright. Los resultados fueron mejores de lo esperado. Para este ejemplo se han calculado 4 rutas. Los puntos verde oscuro denotan el inicio de cada ruta. En la esquina superior derecha se puede ver una leyenda de las rutas calculadas y sus propiedades. Luego de hacer varias pruebas, Clarke and Wright ofrecía de manera consistente rutas eficientes con tiempos de cálculo extremadamente rápidos. El cálculo era instantáneo incluso para instancias del problema con varios cientos de paradas.

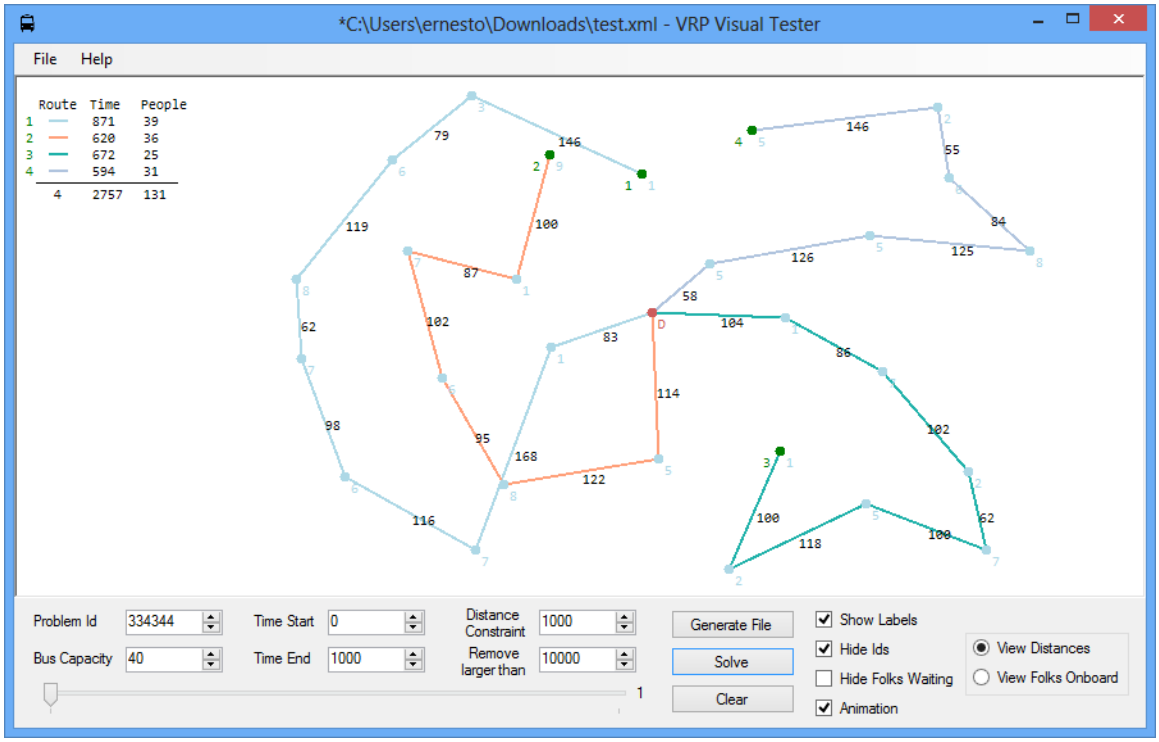


ILUSTRACIÓN 4.5 RUTAS CALCULADAS POR CLARKE AND WRIGHT

## 4.4 OPTIMIZACIÓN MEDIANTE BRANCH AND BOUND

Tras presentar los resultados obtenidos al resto del equipo, a todos nos parecía que la distribución de nodos en rutas estaba aceptable pero que muchas de las rutas podían ser recorridas en mejor orden. Ese sería el próximo paso. Queríamos utilizar la agrupación en rutas calculada por Clarke and Wright y luego optimizar el recorrido de cada ruta.

El nuevo problema era bastante más simple que un VRP, ya que optimizar el recorrido a realizar por un grupo de nodos se parece mucho al problema del viajero (TSP). Específicamente es un TSP "abierto" en el que se puede comenzar en cualquier nodo y hay que terminar en el *depot*.

Como parte de la investigación realizada para el VRP fue inevitable encontrarnos con varias técnicas de solución exitosas para el TSP. Específicamente la técnica de branch and bound en muchos casos sigue siendo el estado del arte en cuanto a calcular soluciones exactas al TSP.

El método de branch and bound es tan potente y genérico que me pareció conveniente hacer un motor de cálculo aparte que tuviese las bases para implementar un algoritmo de este tipo para resolver cualquier problema de optimización. En los anexos BnBSolver.cpp y BnBNode.h se pueden ver los aspectos más importantes de dicho proyecto. La clase BnBNode sirve como clase base a los nodos del árbol de estados del problema de optimización que se quiera resolver. Como se puede apreciar en el código,

los métodos de calcular cota mínima y cota máxima se han definido como abstractos ya que estos son completamente dependientes del problema a resolver.

Según comentamos anteriormente en la sección de branch and bound, el verdadero desafío de una implementación que utilice esta técnica es encontrar un método para calcular cotas mínimas lo más ajustadas posible. Además habíamos visto que una de las mejores maneras de encontrar cotas mínimas era *relajando* el problema. Veamos cuales son las restricciones del problema que queremos resolver a ver como lo podemos *relajar*.

Queremos resolver un TSP "abierto". En otras palabras, dado un grafo completo $G$ queremos seleccionar un subconjunto de aristas que formen un árbol *cadena* que termine en el *depot*. La Ilustración 4.6 muestra a qué nos referimos por árbol *cadena*.


ILUSTRACIÓN 4.6 ÁRBOL CADENA

El problema de encontrar el árbol recubridor mínimo (MST) es ampliamente conocido y cuenta con una gran variedad de algoritmos simples que lo resuelven de manera eficiente. La restricción que le faltaría a este problema para ser un TSP "abierto" es que el árbol tenga forma de *cadena*. Ésta, por tanto, es una perfecta *relajación* para nuestro problema ya que podemos escoger entre una colección de algoritmos conocidos para calcular nuestra cota mínima. Dicho de otra forma, el mejor árbol recubridor de un grafo tendrá necesariamente un costo menor o igual que el mejor árbol recubridor entre los que tienen forma de *cadena*, por lo que el primero es cota mínima del segundo.

El algoritmo escogido para calcular la cota mínima mencionada anteriormente fue el algoritmo de Prim ya que es ampliamente conocido y de fácil implementación. Para calcular cotas máximas se implementó un simple algoritmo *greedy* que comienza desde el *depot* y va en sentido contrario, tomando siempre la mejor opción a la vista. Obviamente, la solución ofrecida por el algoritmo *greedy* será siempre mayor o igual que la mejor solución existente, por lo que puede utilizarse como cota máxima. Para ver detalles sobre estas dos implementaciones el lector puede referirse al anexo RouteBnBNode.cpp.

La Ilustración 4.7 muestra cómo han quedado las rutas calculadas por Clarke and Wright una vez que cada una de ellas ha sido optimizada mediante branch and bound. Como se puede apreciar, la cantidad de rutas y los nodos pertenecientes a cada una se han mantenido constantes y solo se ha modificado el orden en que se recorren éstos. Branch and bound además nos asegura que esta forma de recorrer cada ruta es *óptima*, o sea, que es imposible encontrar un recorrido mejor.

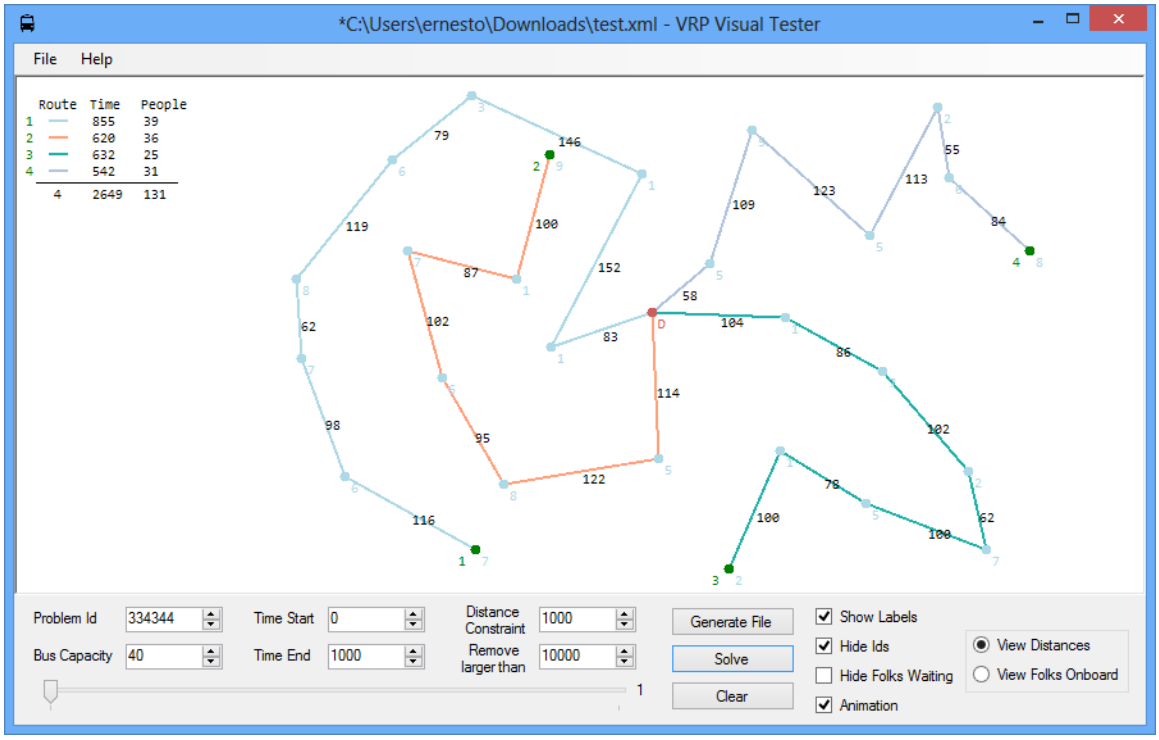


ILUSTRACIÓN 4.7 OPTIMIZACIÓN DE RUTAS MEDIANTE BRANCH AND BOUND

El tiempo de cálculo de optimizar una ruta mediante Branch and bound es dependiente del problema. Por lo general funciona de manera instantánea para rutas de menos de 50 nodos y crece exponencialmente a partir de ese punto.

## 4.5 SOLUCIONES SUCESIVAS MEDIANTE TABUROUTE

Con los resultados obtenidos hasta ahora, ya teníamos un producto que mostrar. Hacer presentaciones, obtener realimentación de los clientes, etc. Se puede decir que habíamos implementado una heurística clásica optimizada, y como tal, nuestro algoritmo ofrecía un resultado decente en un tiempo de cálculo casi instantáneo. Quizás era ese el problema.

La mayoría de los productos de optimización para entornos empresariales son utilizados de manera similar. Las características del problema a optimizar, ya sean personas a recoger o turnos de trabajo a cubrir, suelen cambiar con muy poca frecuencia. En algunos casos cambian cada mes, en otros cada año. Como resultado el cliente casi siempre está dispuesto a comprar un sistema que tarde horas, o incluso días calculando, con tal de obtener mejores soluciones ya que éstas implican mayores ahorros.

Nuestro producto ofrecía soluciones instantáneas, pero éstas estaban alejadas de la solución *óptima* en un 15%. No sería sincero afirmar que con horas extras de cálculo no podíamos reducir ese margen. Necesitábamos un salto en la complejidad de los algoritmos utilizados.

Entre otras cosas, las *meta-heurísticas* han sido tan populares porque se ajustan perfectamente a este tipo de requisitos del entorno empresarial. Podríamos simplificarlo así.

- Los métodos de solución exacta dicen: Hasta que no encuentre una solución óptima no terminaré la ejecución.
- Las heurísticas clásicas dicen: Terminaré la ejecución en 1 segundo sin importar qué pase. Ofreceré la mejor solución que encuentre en ese tiempo.
- Las meta-heurísticas dicen: Dime qué tiempo tienes disponible para calcular y haré lo mejor que pueda.

La situación requería una *meta-heurística*. Durante nuestro recorrido por la literatura vimos que desde 1990 habían sido propuestas varias muy prometedoras. Las *meta-heurísticas* se caracterizan por ser complejas e involucrar muchos detalles de implementación. Es por eso que estudiarlas todas para decidir cuál utilizar nos es precisamente factible en un entorno empresarial. En su lugar hay que llevarse una idea general de cómo funciona cada una y recopilar tantos resultado experimentales y tablas comparativas como se pueda. Son éstas las que dan una idea aproximada de si el método a utilizar es apropiado.

Luego de haber hecho el estudio, *Taburoute* nos convenció a casi todos. Parecía simple de comprender, utilizaba ideas ingeniosas y los resultados experimentales eran increíblemente buenos. El próximo paso sería buscar las fuentes originales que propusieron *Taburoute* por primera vez para extraer de éstas todos los detalles de implementación necesarios. Estamos hablando, por supuesto, del paper publicado por Gendreau, Hertz y Laporte en 1994 [8], y del paper que sirve de base a éste, publicado dos años antes por los mismos autores [9].

### *4.5.1 PREÁMBULO Y NOTACIÓN DE TABUROUTE*

Ya que *Taburoute* fue finalmente escogido e implementado en nuestro proyecto y pasó a convertirse en una parte medular del cálculo, creo que merece una sección protagónica en esta memoria. A continuación procederemos a explicar cómo funciona.

Utilizaremos la siguiente notación. Una solución es un conjunto $S$ de $m$ rutas $R_1, \dots, R_m$. Cada ruta puede ser representada como $R_r = (v_{r_1}, v_{r_2}, \dots, v_0)$ (recordar que son rutas abiertas), y cada vértice $v_i$ $(i \geq 1)$ pertenece a exactamente una ruta. Estas rutas pueden ser válidas o inválidas con respecto a las restricciones de capacidad o distancia. Por conveniencia diremos que $v_i \in R_r$ si el vértice $i$ pertenece a la ruta $r$, y $(v_i, v_j) \in R_r$ si el arco $(i, j)$ pertenece a la ruta $r$.

Para cualquier solución válida nuestra función objetivo será:

$$F_1(S) = \sum_r \sum_{(v_i, v_j) \in R_r} c_{ij}$$

Adicionalmente, aquellas soluciones (válidas o no) tendrán una segunda función objetivo:

$$F_2(S) = F_1(S) + \alpha Q_e + \beta L_e$$

Donde $\alpha$ y $\beta$ son parámetros del algoritmo, mientras que $Q_e$ y $L_e$ denotan cuánto la solución está violando las restricciones de capacidad o distancia, respectivamente. Nótese que si la solución es válida las funciones objetivo $F_1(S)$ y $F_2(S)$ darán el mismo valor, sin embargo para las soluciones inválidas $F_2(S)$ incorpora dos términos de penalización por el exceso de capacidad y el exceso de distancia. En todo momento $F_1^*$ y $F_2^*$ denotarán los mejores valores encontrados hasta ahora de $F_1(S)$ y $F_2(S)$. De manera similar, $S_1^*$ denotará la mejor solución encontrada que sea válida (o sea, que no viole ninguna restricción) y $S_2^*$ la mejor solución encontrada en general, ya sea válida o inválida con respecto a las restricciones de capacidad y distancia.

Primero vamos a describir un procedimiento de búsqueda medular a *Taburoute* que llamaremos Search(P). Este procedimiento parte de una solución inicial e intenta mejorarla utilizando una técnica basada en *Tabu search.* Este procedimiento además, reutiliza dos heurísticas propuestas por los mismos autores para darle solución al TSP llamadas GENI y US. La primera recibe su nombre de *Generalized Insertion* y se utiliza para determinar una ruta eficiente que recorra un grupo dado de vértices. La segunda recibe su nombre de *Unstringing and Stringing* y se utiliza para re-optimizar una ruta ya formada utilizando sucesivas eliminaciones y reinserciones de vértices. Dichas eliminaciones y reinserciones (y todas las que se realicen durante *Taburoute*) se harán utilizando los métodos de *Stringing* y *Unstringing* propuestos también por estos autores. Los métodos GENI, US, *Stringing* y *Unstringing* los veremos más adelante.

El procedimiento de búsqueda Search(P) está regido por un vector de parámetros:

$$P = (W, q, p_1, p_2, \theta_{min}, \theta_{max}, g, h, n_{max})$$

Donde cada uno denota lo siguiente:

- $W$: es un subconjunto no vacío de $V \setminus \{0\}$ que contiene los vértices que tienen permitido moverse durante el proceso de búsqueda.
- $q$: la cantidad de vértices de W que se toman en cada iteración como candidatos a ser reinsertados en otra ruta.
- $p_1$: la ruta en la que un vértice $v$ es reinsertado debe contener al menos 1 de sus $p_1$ vecinos más cercanos.
- $p_2$: el tamaño de las vecindades utilizadas en GENI.
- $\theta_{min}, \theta_{max}$: cotas mínima y máxima a la cantidad de iteraciones que una movida permanece como *tabú.*
- $g$: factor de penalización que se utiliza para los vértices que se mueven con mucha frecuencia.
- $h$: cantidad de iteraciones consecutivas sin reajustar los parámetros $\alpha$ y $\beta$.

- $n_{max}$: máxima cantidad de iteraciones consecutivas que pueden transcurrir sin mejora, antes de terminar la ejecución.

### *4.5.2 PSEUDOCÓDIGO DEL PROCEDIMIENTO SEARCH(P)*

**Paso 0 (Inicialización)**: Se inicializa el contador de iteraciones $t = 1$. Se vacía la lista de movidas *tabú.*

**Paso 1 (Selección de vértices)**: Se seleccionan aleatoriamente $q$ vértices del conjunto $W$.

**Paso 2 (Considerar todas las movidas candidatas)**: Para cada vértice seleccionado $v$ y cada ruta $R_r$ que contenga al menos 1 de sus $p_1$ vecinos más cercanos, se ejecutan los siguientes pasos:

a) Elimina $v$ de su ruta original utilizando *Unstringing* y calcula cual sería el costo de su reinserción en $R_r$ utilizando *Stringing* con tamaño de vecindad $p_2$. Se guarda en $S'$ la solución que se obtendría o al menos los campos más importantes de esta.
b) Si la movida de $v$ a $R_r$ es *tabú* en la iteración actual, se descarta excepto en el caso de que $S'$ mejore alguno de los costos $F_1^*$ o $F_2^*$.
c) Si $S'$ no fue descartada, el costo por el cual se va a evaluar se define como $F(S') = F_2(S') + \Delta_{max}\sqrt{m}gf_v$, donde $\Delta_{max}$ es la mayor diferencia absoluta de costo encontrada en soluciones consecutivas y $f_v$ es la frecuencia con la que el vértice $v$ ha sido movido (cantidad de veces que ha sido movido dividida por $t$).

**Paso 3 (Identificar la mejor movida)**: La solución vecina con menor valor de $F(S')$ se almacena en $S''$.

**Paso 4 (Moverse a la siguiente solución)**: La solución obtenida en el Paso 3 no siempre se toma. A veces es mejor re-optimizar la solución actual utilizando US. Específicamente si la solución actual es mejor que $S'$ y US no fue utilizado en la última iteración, se descarta $S'$ y se aplica la heurística de reoptimización US a la solución actual. En caso contrario la solución actual $S = S'$.

**Paso 5 (Actualización de variables)**: Si se utilizó la movida en el Paso 4, reinsertar el vértice $v$ en su ruta original es declarado *tabú* hasta la iteración $t + \theta$ donde $\theta$ es un entero escogido aleatoriamente en el rango $[\theta_{min}, \theta_{max}]$. Adicionalmente se actualizan las variables $F_1^*,\ F_2^*,\ S_1^*,\ S_2^*,\ \Delta_{max},\ m$ y $f_v$. Se incrementa el número de iteración $t = t + 1$.

**Paso 6 (Ajuste de parámetros)**: Si el número de iteración $t$ es múltiplo de de $h$ se procede a re-ajustar los factores de penalización $\alpha$ y $\beta$. Si las últimas $h$ iteraciones fueron todas válidas con respecto a la restricción de capacidad $\alpha = \alpha/2$, si fueron todas inválidas $\alpha = 2\alpha$. De manera análoga se procede con el factor $\beta$ y la restricción de distancia. El objetivo es que cada $h$ iteraciones se analicen tanto soluciones válidas como inválidas. Es de esa frontera que salen las mejores soluciones.

**Paso 7 (Condición de parada)**: Si los valores de $F_1^*$ y $F_2^*$ no han sido modificados en las últimas $n_{max}$ iteraciones, se termina la ejecución. De lo contrario se regresa al Paso 1.

### *4.5.3 PSEUDOCÓDIGO DEL ALGORITMO TABUROUTE*

Ya que hemos visto en qué consiste el procedimiento de búsqueda, podemos pasar a describir el algoritmo en sí. Inicialmente se generan varias soluciones iniciales y se ejecuta sobre ellas una búsqueda limitada. La mejor candidata luego es escogida para ejecutar una búsqueda más profunda. Finalmente se ejecuta una tercera búsqueda para acercar la solución al mínimo local más cercano antes de finalizar.

**Paso 0 (Inicialización)**: Se inicializan los valores de $\alpha = 1$ y $\beta = 1$. Adicionalmente se inicializa $F_1^* = +\infty$ y $F_2^* = +\infty$.

**Paso 1 (Soluciones iniciales)**: Se repiten los siguientes pasos $\lambda$ veces:

a) Se crea una ruta que pase por todos los vértices del problema utilizando la heurística GENI de solución al TSP. A esta heurística se le dan los vértices en un orden aleatorio.
b) Se corta la ruta creada en tantas rutas consecutivas como sea necesario para que ninguna viole las restricciones de capacidad o distancia.
c) Se invoca el método Search($P_1$) sobre la solución creada, con un vector de parámetros conservador.

**Paso 2 (Mejora de la solución)**: La mejor solución encontrada en el Paso 1 estará almacenada en $S_2^*$. Se toma ésta de partida para realizar una segunda búsqueda Search($P_2$) con un vector de parámetros más agresivo.

**Paso 3 (Intensificación de la solución)**: Se invoca el proceso de búsqueda Search($P_3$) con un vector de parámetros diseñado para que la solución se aproxime al mínimo local más cercano antes de finalizar la ejecución.

### *4.5.4 SELECCIÓN DE PARÁMETROS*

Una gran parte del arte de las *meta-heurísticas* consiste en sintonizar los parámetros de búsqueda de manera apropiada. A continuación comentamos cómo la literatura recomienda que se seleccionen los parámetros para *Taburoute* y cómo en definitiva los seleccionamos para nuestro producto.

- El valor de $\lambda$ suele definirse como $\frac{\sqrt{|V|}}{2}$.
- Los siguientes parámetros se mantienen contantes en las 3 fases de búsqueda, por lo que son comunes a $P_1$, $P_2$ y $P_3$:
  - $g = 0.01$
  - $\theta_{min} = 5$
  - $\theta_{max} = 10$
  - $h = 10$
  - $p_2 = 5$

- $p1 = \max(k, p2)$ donde $k$ es la longitud de la ruta que contiene al vértice movido.

- Parámetros específicos de $P_1$:
  - $W = V \setminus \{0\}$
  - $q = 5m$ donde $m$ es la cantidad de rutas. Se ha estudiado que éste es el valor más conveniente para aumentar la probabilidad de que en cada iteración se considere mover al menos un vértice de cada ruta.
  - $n_{max} = |V|$
- Parámetros específicos de $P_2$:
  - $W = V \setminus \{0\}$
  - $q = 5m$
  - $n_{max} = 50|V|$
- Parámetros específicos de $P_3$:
  - $W = \{\frac{|V|}{2}\ vértices\ que\ tenga\ mayor\ f_v\}$: la búsqueda se concentra en mover sólo aquellos vértices "controvertidos", o sea, aquellos que el algoritmo ha movido con mayor frecuencia.
  - $q = \frac{|V|}{2}$: se toman todos los vértices de $W$ en cada iteración.
  - $n_{max} = |V|$

Como se ha podido observar una gran parte de *Taburoute* depende de las heurísticas anteriores propuestas por estos autores, por lo que no podemos dejar de comentar en qué consisten y cómo las adaptamos a nuestra versión de VRP con rutas "abiertas".

### *4.5.5 INSERCIÓN DE VÉRTICES MEDIANTE STRINGING*

Cuando tenemos que insertar un vértice nuevo en una ruta existente nos surgen las siguientes preguntas. ¿Dónde ubicamos el nuevo vértice? ¿Cómo reordenamos la ruta incrementada? Simplemente probar ubicando el nuevo vértice entre cada par de vértices consecutivos puede parecer una buena idea, sin embargo no lo es. Hay una gran variedad de casos en los que la mejor ruta tras insertar un vértice es completamente diferente a la ruta original. Este problema no es simple, y el reto consiste en no probar todas las maneras posibles sino solo aquellas que sean más prometedoras. El método *Stringing* es una propuesta de cómo afrontar este problema. Es mi opinión que en la fortaleza de método *Stringing* y su análogo *Unstringing* radica el valor del algoritmo *Taburoute* e incluso de su predecesor para el TSP, GENI-US.

Veamos entonces qué propone *Stringing.* Se tiene la ruta $R_r = (v_{r_1}, v_{r_2}, \dots, v_{r_{|R_r|}})$. Se quiere insertar un vértice en ésta denotado simplemente por $v$. Supongamos que $v_{i+1}$ y $v_{i-1}$ son los vértices sucesor y predecesor del vértice $v_i$ dentro de la ruta $R_r$ y que la relación $v_i < v_j$ significa que el vértice $v_i$ aparece primero en la ruta que el vértice $v_j$. Adicionalmente $N_p(v_i)$ denotará el conjunto de los $p$ vértices más cercanos a $v_i$. *Stringing* propone que la ruta sea reconectada de alguna de las siguientes dos maneras.

*Stringing* Tipo 1:

Para todo $v_i, v_j, v_k \in V \setminus \{0\}$ tales que $v_i < v_j < v_k$, $v_i, v_j \in N_p(v)$ y $v_k \in N_p(v_{i+1})$, considerar la ruta que se obtendría tras eliminar los arcos $(v_i, v_{i+1})$, $(v_j, v_{j+1})$ y $(v_k, v_{k+1})$, e insertar los arcos $(v_i, v)$, $(v, v_j)$, $(v_{i+1}, v_k)$ y $(v_{j+1}, v_{k+1})$. La Ilustración 4.8 muestra cómo quedaría la nueva ruta con el vértice $v$ insertado.

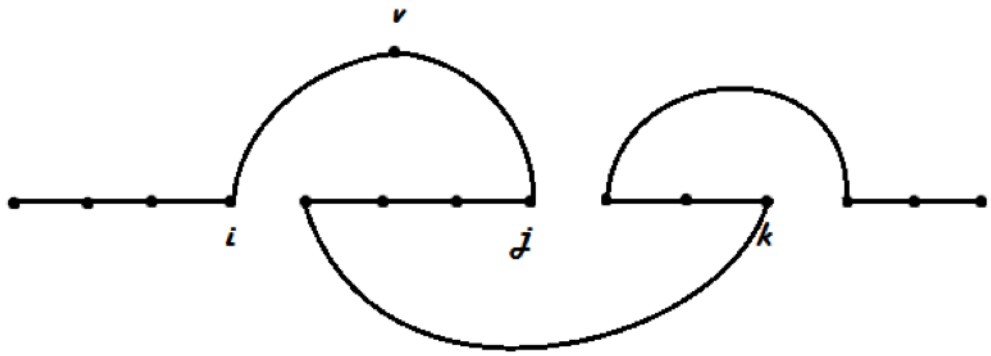


ILUSTRACIÓN 4.8 *STRINGING* TIPO 1

*Stringing* Tipo 2:

Para todo $v_i, v_j, v_k, v_l \in V \setminus \{0\}$ tales que $v_i < v_{l-1} < v_j < v_{k-1}$, $v_i, v_j \in N_p(v)$, $v_l \in N_p(v_{j+1})$ y $v_k \in N_p(v_{i+1})$, considerar la ruta que se obtendría tras eliminar los arcos $(v_i, v_{i+1})$, $(v_j, v_{j+1})$, $(v_{l-1}, v_l)$ y $(v_{k-1}, v_k)$, e insertar los arcos $(v_i, v)$, $(v, v_j)$, $(v_l, v_{j+1})$, $(v_{k-1}, v_{l-1})$ y $(v_{i+1}, v_k)$. La Ilustración 4.9 muestra cómo quedaría la nueva ruta con el vértice $v$ insertado.

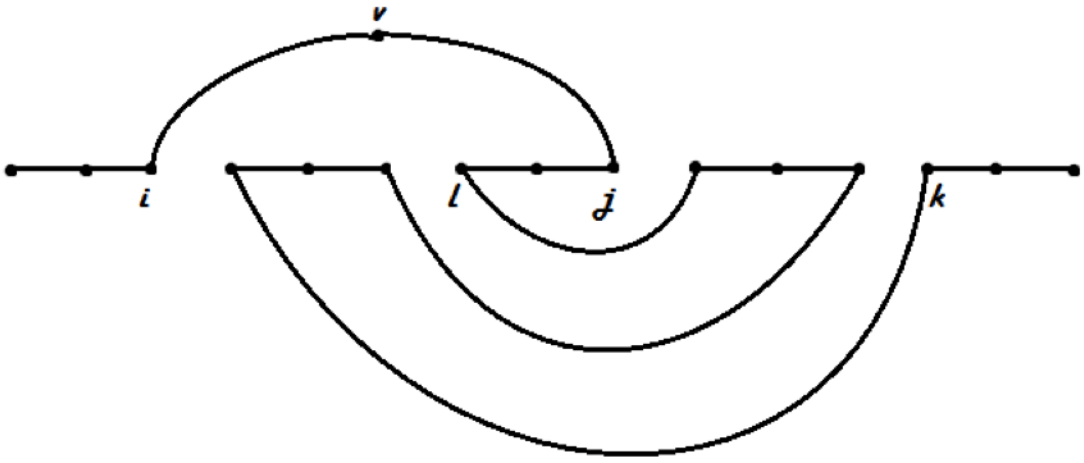


ILUSTRACIÓN 4.9 *STRINGING* TIPO 2

### 4.5.6 *Eliminación de vértices mediante Unstringing*

De manera análoga a *Stringing*, éste es un método que propone una manera de reconectar una ruta tras haber eliminado alguno de sus vértices y al igual que *Stringing* ofrece dos tipos de reconexiones que se deberían considerar.

*Unstringing* Tipo 1:

Sea $v_i$ el vértice que se quiere eliminar de la ruta. Para todo $v_j, v_k \in V \setminus \{0\}$ tales que $v_i < v_k < v_j$, $v_j \in N_p(v_{i+1})$ y $v_j \in N_p(v_{i-1})$, considerar la ruta que se obtendría tras eliminar los arcos $(v_{i-1}, v_i)$, $(v_i, v_{i+1})$, $(v_k, v_{k+1})$ y $(v_j, v_{j+1})$, e insertar los arcos $(v_{i-1}, v_k)$, $(v_{i+1}, v_j)$ y $(v_{k+1}, v_{j+1})$. La Ilustración 4.10 muestra cómo quedaría la nueva ruta tras haber eliminado el vértice $v_i$.

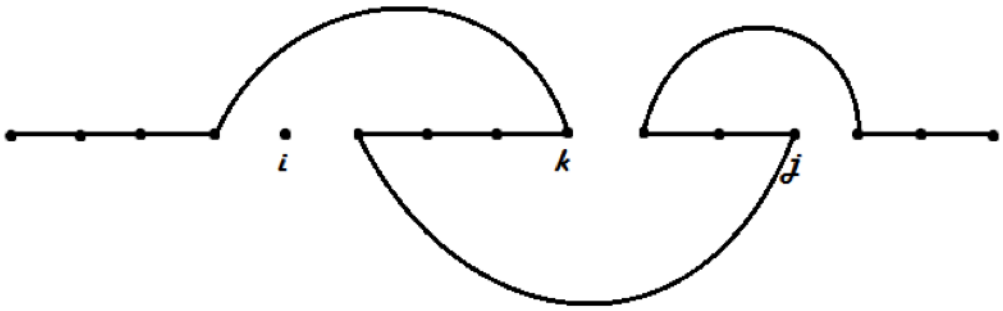


Ilustración 4.10 *Unstringing* tipo 1

*Unstringing* Tipo 2:

Sea $v_i$ el vértice que se quiere eliminar de la ruta. Para todo $v_j, v_k, v_l \in V \setminus \{0\}$ tales que $v_i < v_{j-1} < v_l < v_k$, $v_j \in N_p(v_{i+1})$, $v_k \in N_p(v_{i-1})$ y $v_l \in N_p(v_{k+1})$, considerar la ruta que se obtendría tras eliminar los arcos $(v_{i-1}, v_i)$, $(v_i, v_{i+1})$, $(v_{j-1}, v_j)$, $(v_l, v_{l+1})$ y $(v_k, v_{k+1})$, e insertar los arcos $(v_{i-1}, v_k)$, $(v_{l+1}, v_{j-1})$, $(v_{i+1}, v_j)$ y $(v_l, v_{k+1})$. La Ilustración 4.11 muestra cómo quedaría la nueva ruta tras haber eliminado el vértice $v_i$.

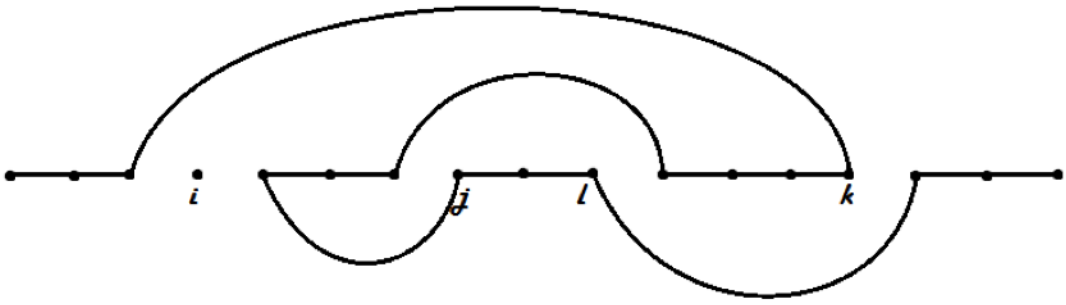


Ilustración 4.11 *Unstringing* tipo 2

### 4.5.7 *Pseudocódigo de GENI*

GENI es la heurística propuesta por estos autores para resolver el TSP que en *Taburoute* es utilizada para encontrar soluciones iniciales.

**Paso 1 (Inicialización)**: Crear una sub-ruta inicial seleccionando tres vértices aleatorios. Inicializar las vecindades de todos los vértices (conjunto de sus $p$ vértices más cercanos).

**Paso 2 (Inserción de vértice)**: Aleatoriamente seleccionar un vértice $v$ que no pertenezca aún a la sub-ruta. Insertar el vértice $v$ en la sub-ruta utilizando el método *Stringing*.

**Paso 3 (Condición de parada)**: Si todos los vértices son parte de la sub-ruta terminar la ejecución. De lo contrario regresar al Paso 2.

### 4.5.8 *Pseudocódigo de US*

Adicionalmente, los autores también propusieron una heurística de reoptimización de ruta. Esta heurística se utiliza en el Paso 4 del procedimiento Search(P) de *Taburoute*. La heurística consiste en sucesivas eliminaciones y reinserciones de vértices en la ruta, de ahí recibe su nombre *Unstringing and Stringing*.

**Paso 1 (Inicialización)**: Se recibe la ruta $r = (v_1, v_2, \dots, v_{|r|})$. Se inicializa $t = 1$.

**Paso 2 (US)**: Se elimina el vértice $v_t$ de la ruta $r$ utilizando el método *Unstringing.* Luego se reinserta utilizando el método *Stringing*.

**Paso 3 (Actualización)**: Si el costo de $r$ fue mejorado en el Paso 2, se comienza desde el principio $t = 1$. De lo contrario $t = t + 1$.

**Paso 4 (Condición de parada)**: Si $t > |r|$, terminar la ejecución. De lo contrario regresar al Paso 2.

### 4.5.9 *Resultado de Taburoute*

La implementación de *Taburoute* resultó ser más complicada y extensa de lo inicialmente previsto. En total ocupó el tiempo de una persona a tiempo parcial durante dos meses y llegó crecer hasta unas 2000 líneas de C++. Hubo muchos obstáculos de implementación en los que no hemos profundizado, pero que también llevaron investigación como:

- Escoger $q$ vértices aleatorios del conjunto $W$ en $O(|W|)$, implementado en el método $RandomlySelect()$
- Seleccionar los $k$ vértices más cercanos a $v$ de un conjunto de tamaño $n$, en $O(n \log k)$, implementado en el método $ClosestVerticesFrom()$.
- Calcular el costo de la ruta que se forma luego de *Stringing* o *Unstringing* en $O(1)$. Para esto se utilizaron *arrays* acumulativos de costos.

Para ver estos y otros detalles de implementación el lector puede referirse a *Taburoute.cpp* en los anexos. La Ilustración 4.12 muestra la solución que ofreció *Taburoute* al caso de prueba que hemos estando utilizando hasta ahora.

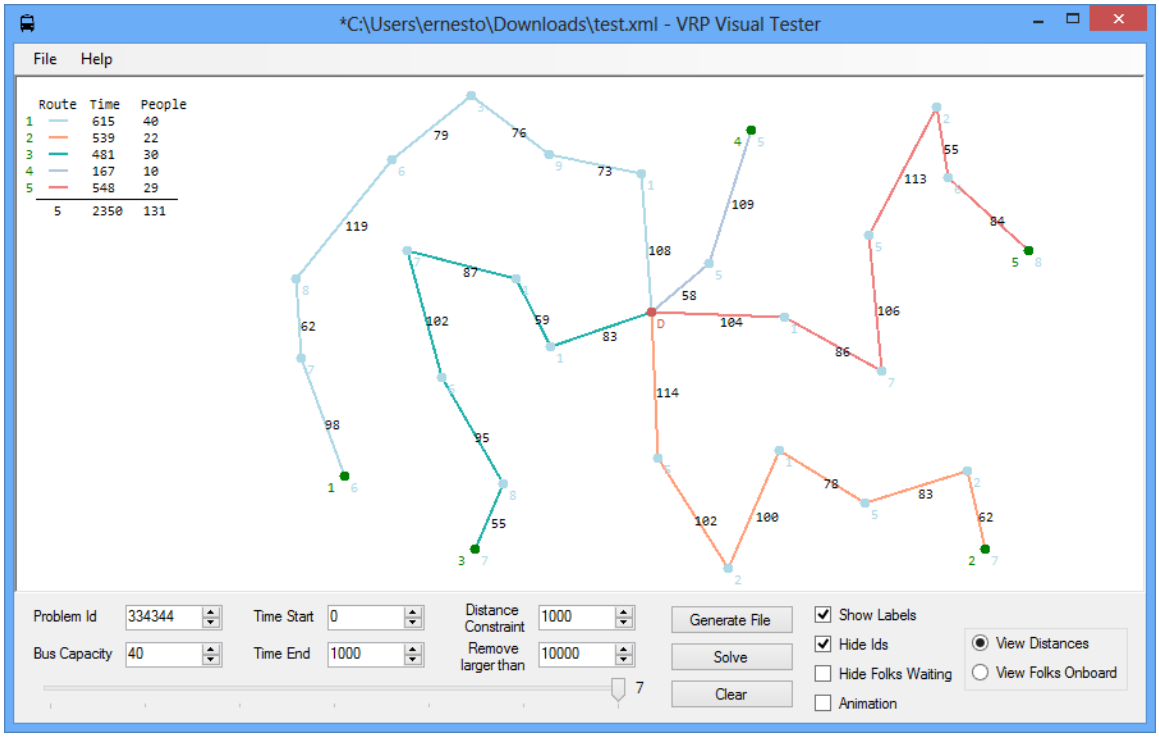


ILUSTRACIÓN 4.12 SOLUCIÓN CALCULADA POR TABUROUTE

Como se puede apreciar, hay una diferencia significativa entre la solución calculada por *Taburoute* y las soluciones calculadas anteriormente y para un problema tan pequeño la diferencia en tiempo de ejecución es aún despreciable.

# 5 RESULTADOS Y EVALUACIÓN

Para hacer una comparación de resultados hemos generado 6 casos de prueba (7 si contamos el básico utilizado hasta ahora). Todos asumen que pueden utilizar autobuses de capacidad 40 y que la ruta más larga puede tener a lo sumo 1000 píxeles. Cuentan con paradas homogéneamente distribuidas que tienen entre 1 y 9 personas esperando. El *depot* suele estar centrado, con la excepción del caso 3 en el que el *depot* está desplazado hacia una esquina del plano. Se ha ido incrementando la cantidad de vértices, empezando con 25 y terminando en 200. Pasemos a ver ilustraciones de los resultados ofrecidos para cada caso y comentarlos posteriormente.

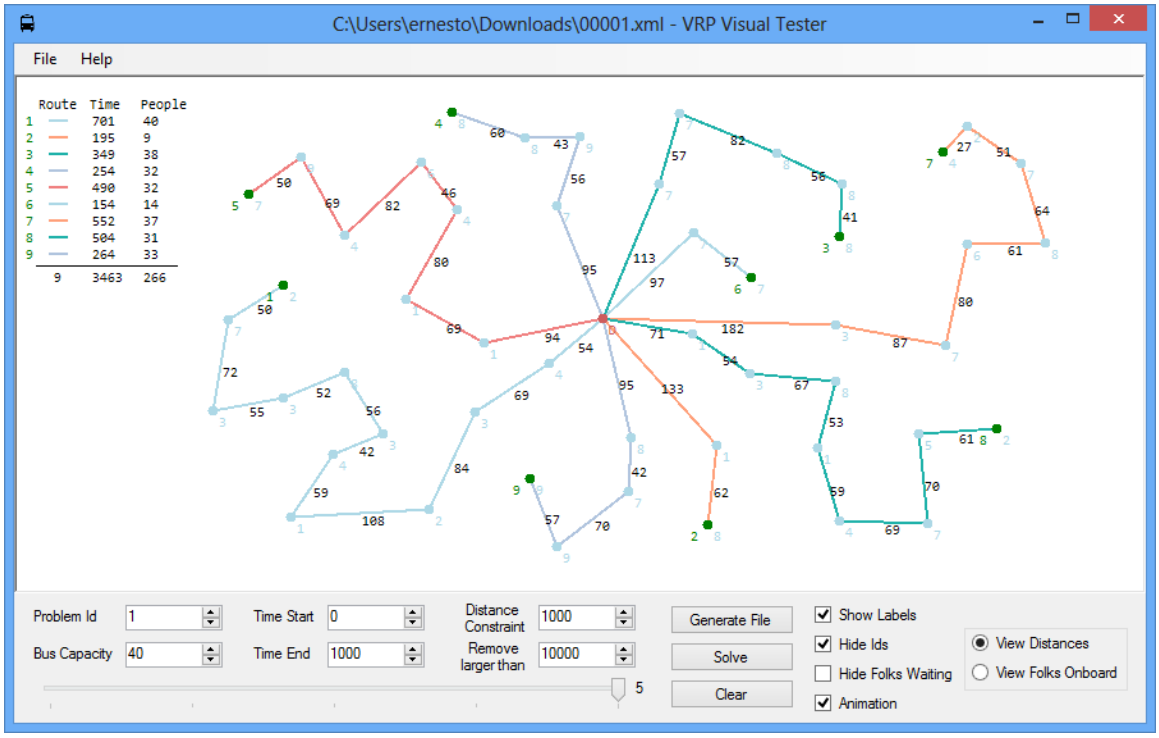


ILUSTRACIÓN 5.1 SOLUCIÓN CALCULADA PARA EL CASO 1

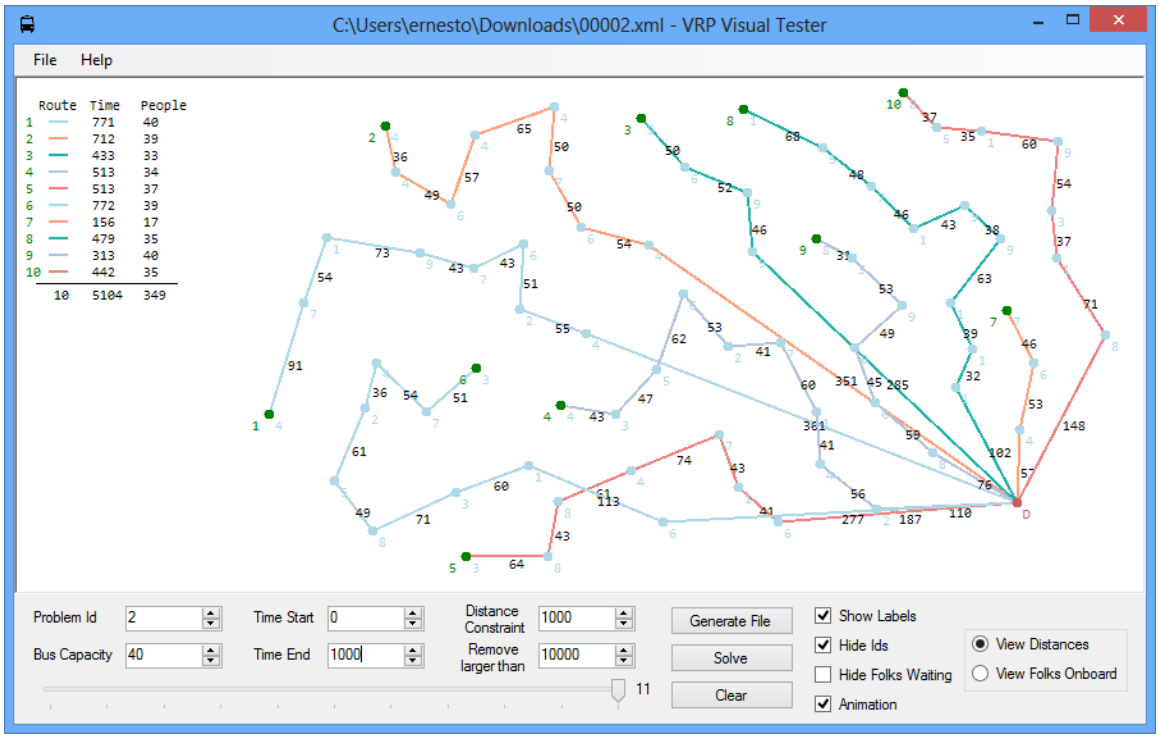


ILUSTRACIÓN 5.2 SOLUCIÓN CALCULADA PARA EL CASO 2

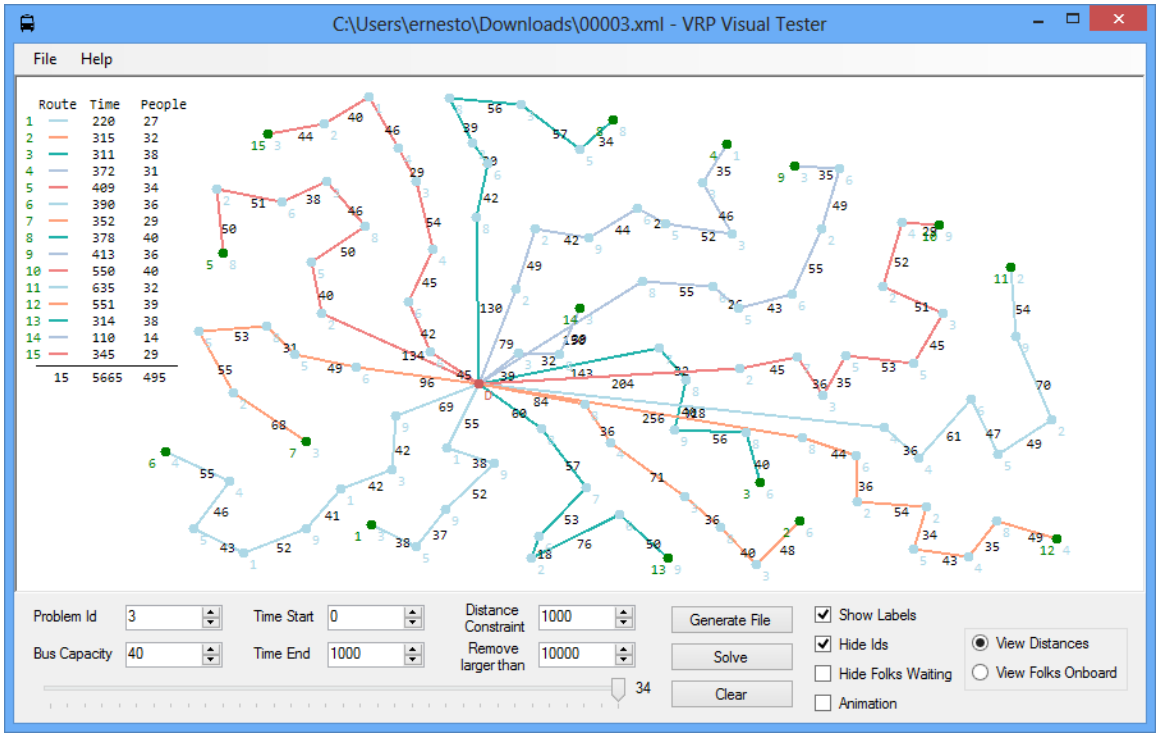


ILUSTRACIÓN 5.3 SOLUCIÓN CALCULADA PARA EL CASO 3

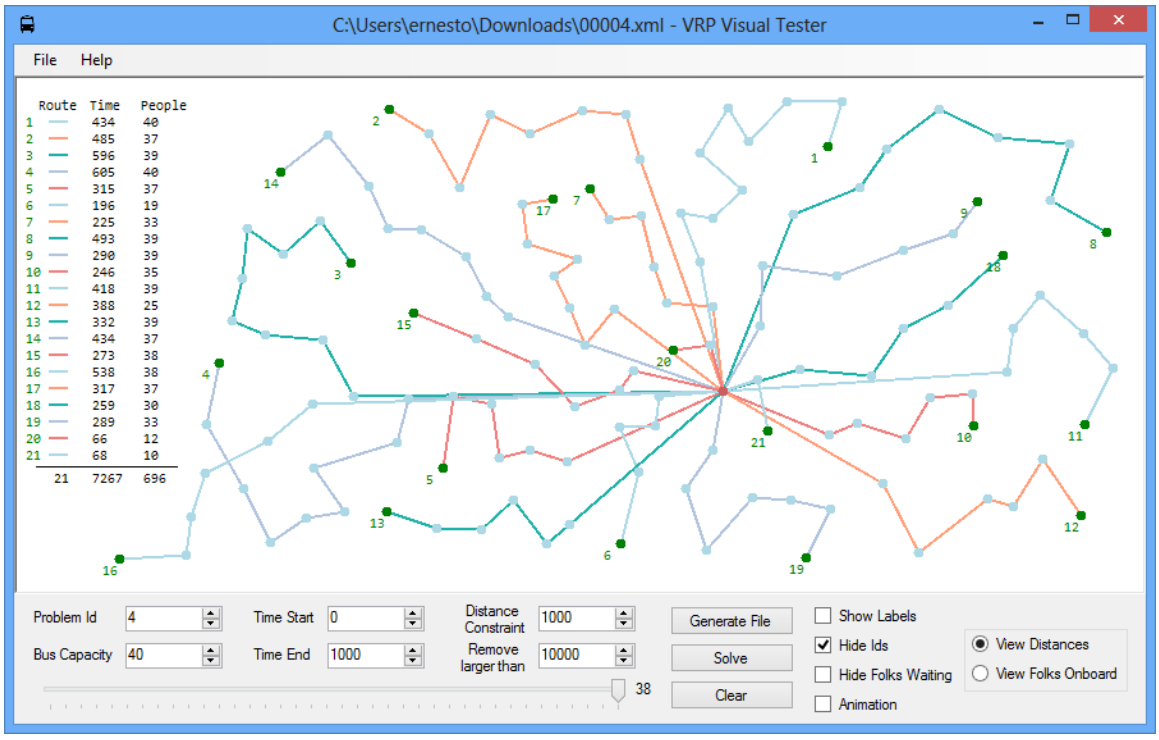


ILUSTRACIÓN 5.4 SOLUCIÓN CALCULADA PARA EL CASO 4

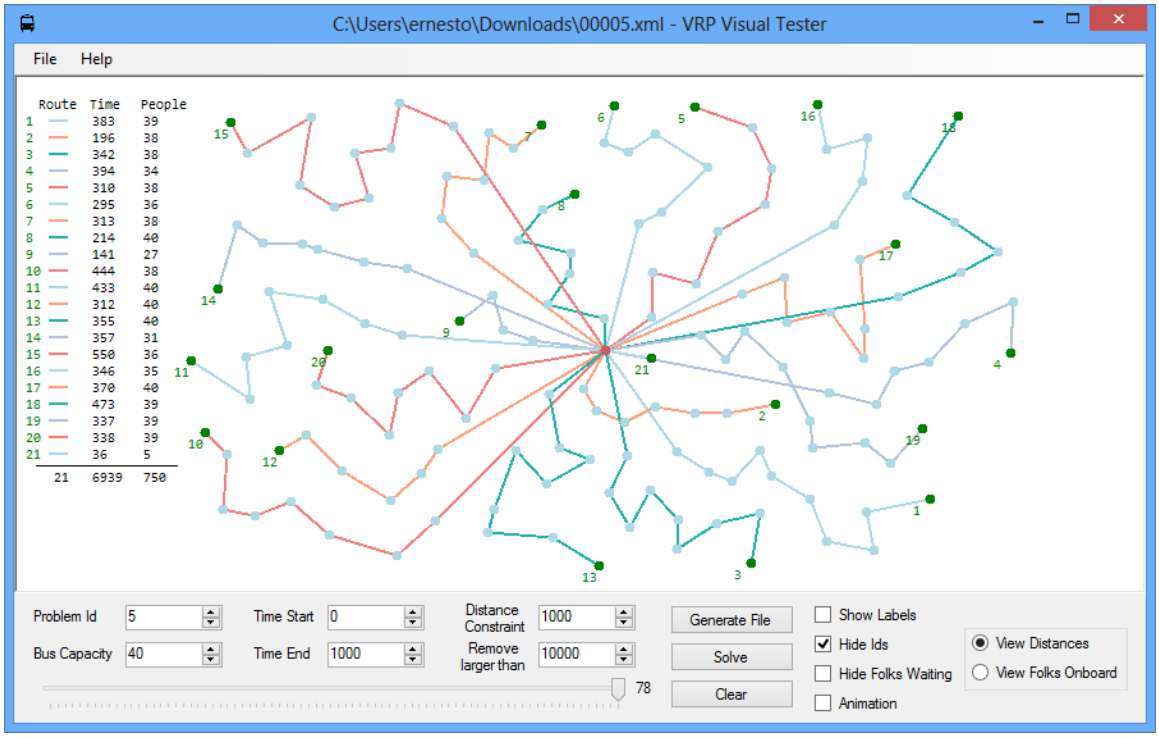


ILUSTRACIÓN 5.5 SOLUCIÓN CALCULADA PARA EL CASO 5

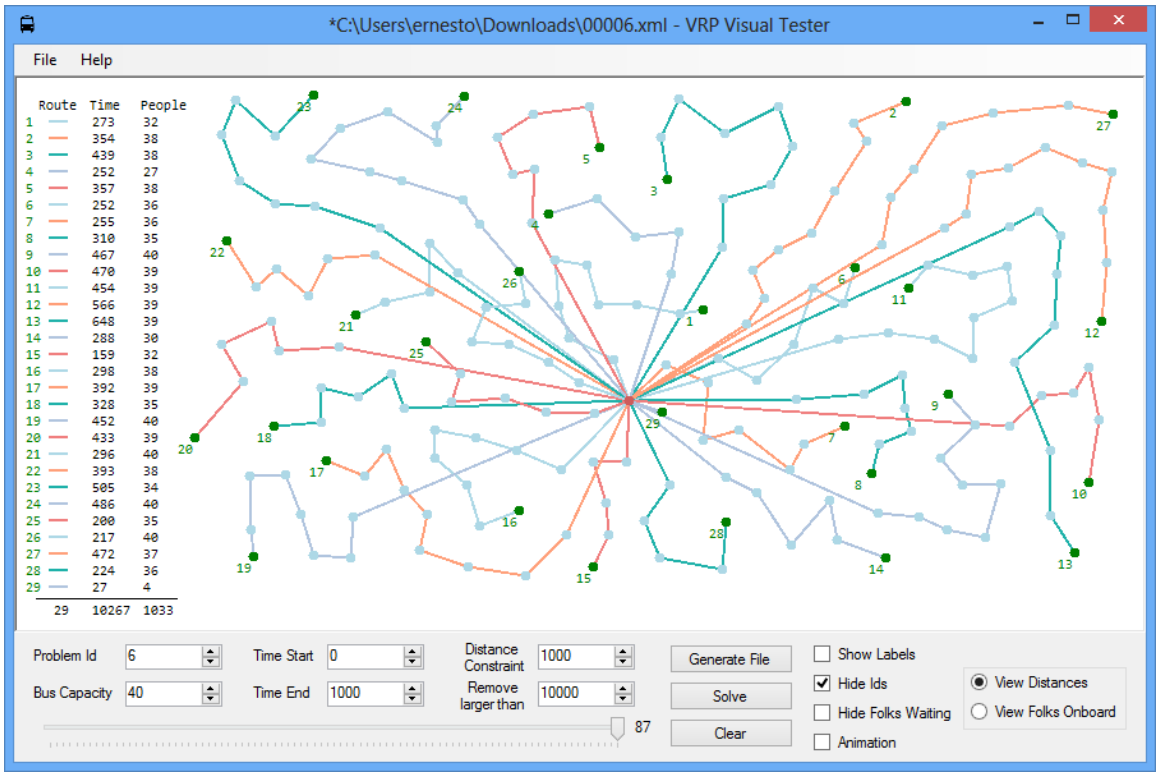


ILUSTRACIÓN 5.6 SOLUCIÓN CALCULADA PARA EL CASO 6

Para el caso 1 de 50 vértices, el producto ha encontrado 5 soluciones. Cada una de estas soluciones podría observarse utilizando la barra deslizadora de abajo. La Ilustración 5.1 muestra la mejor solución encontrada. Ésta tiene 9 rutas que transportan a 266 personas con un costo de 3463 (píxeles). De manera similar se puede leer la solución a los demás casos. A partir del caso 4 se ha desmarcado la casilla "Show Labels" ya que la imagen quedaba demasiado cargada de etiquetas. No obstante, la leyenda de cada solución se puede seguir viendo a la derecha. La Tabla 5.1 muestra una comparación entre los casos de prueba. Nótese que la particularidad del caso 3 hace que la diferencia entre las soluciones calculadas por Clarke and Wright y *Taburoute* sea enorme.

| Identificador del problema | Cantidad de vértices | Solución ofrecida por C&W y B&B | Tiempo de cálculo | Cantidad de soluciones encontradas por Taburoute | Mejor solución encontrada por Taburoute | Tiempo de cálculo |
|---|---|---|---|---|---|---|
| 00000 | 25 | 2649 | 0.0s | 5 | 2350 | 0.0s |
| 00001 | 50 | 3754 | 0.0s | 11 | 3463 | 1.5s |
| 00002 | 70 | 8718 | 0.0s | 13 | 5104 | 4.1s |
| 00003 | 100 | 6150 | 0.0s | 34 | 5665 | 9.2s |
| 00004 | 130 | 7889 | 0.0s | 38 | 7267 | 18.9s |
| 00005 | 150 | 7635 | 0.0s | 78 | 6939 | 33.8s |
| 00006 | 200 | 10865 | 0.1s | 87 | 10267 | 45.3s |

TABLA 5.1 TABLA COMPARATIVA DE LOS CASOS DE PRUEBA

# 6 POSIBLES MEJORAS

Esta sección la vamos a dedicar comentar algunas de las debilidades que hemos encontrado en nuestro producto y cómo podríamos mejorarlas.

## 6.1 PROBLEMA DE ESCALABILIDAD Y SOLUCIÓN

Una de las debilidades más evidentes de nuestro producto es los grandes tiempos de ejecución a partir de cierto tamaño de la entrada. La Tabla 5.1 muestra como los tiempos de ejecución comienzan a crecer rápidamente a partir de 150 vértices. Este problema es bastante común a todos los algoritmos de optimización ya que éstos suelen lidiar con problemas que son *NP-Hard*. Por lo general los algoritmos de optimización se diseñan con un rango de entrada en mente. Si ese rango cambia drásticamente hay que hacer modificaciones en el algoritmo que tendrán repercusiones en la calidad de la solución. Hay dos principales responsables de las demoras en un algoritmo, operaciones del procesador y operaciones de entrada y salida. Veremos cómo lidiar con cada una en orden.

Para mejorar el tiempo de procesamiento se puede recurrir a mejorar el hardware utilizado o introducir concurrencia, pero si la demora es inmanejable necesariamente habrá que modificar o sintonizar la heurística utilizada. Las heurísticas en general suelen ser una negociación entre tiempo de ejecución y calidad de los resultados. En la Tabla 5.1 vemos como nuestra versión optimizada del algoritmo de Clarke and Wright, de orden aproximado $O(n^2)$, ofrece tiempos de cálculo instantáneos para todos los casos de prueba. *Taburoute* por su parte mejora la solución ofrecida por C&W en un 10% como promedio, a cambio de un tiempo de cálculo elevado. Sería necesario encontrar un punto intermedio si el tamaño de la entrada rondara los miles de vértices. En *Taburoute*, como en la mayoría de las *meta-heurísticas*, esto se logra sintonizando los parámetros de entrada al algoritmo, específicamente modificando el vector de parámetros $P$ que sirve de entrada al procedimiento de búsqueda Search(P). Tras comprender este procedimiento uno se percata que modificar los valores de $p_1, p_2, q$ y $n_{max}$ tendría implicaciones directas en el tiempo de ejecución y en la calidad de los resultados ofrecidos.

Para reducir la demora provocada por las operaciones de entrada y salida también contamos con varias opciones. La primera que nos viene a la mente, que es independiente del tipo de entrada y salida que estemos utilizando, sería utilizar concurrencia para este tipo de operaciones. En la versión actual de nuestro producto, las operaciones de entrada y salida se ejecutan en el mismo hilo que el cálculo en sí. Paralelizar estas operaciones sería relativamente simple y tendría implicaciones positivas en el tiempo de ejecución. Adicionalmente, por supuesto, se podría mejorar la tecnología involucrada en las operaciones de entrada y salida. En este caso estamos utilizando lecturas y escrituras a disco duro. Mejoras en el hardware o en el driver (ej. utilizar discos de estado sólido o utilizar mejores sistemas de ficheros respectivamente) podrían reducir el tiempo de ejecución.

## 6.2 PROBLEMAS DE VERSATILIDAD Y SOLUCIONES

En los productos de optimización es recurrente el problema de versatilidad. Las propuestas matemáticas salidas de entornos académicos funcionan de maravilla en entornos controlados donde las restricciones están claras y prefijadas. Sin embargo el mundo real es diferente. La realidad es cambiante y los clientes tienen requisitos nuevos cada mes. Es por eso que algunas plataformas, como ILOG de IBM, han sido tan exitosas en el mundo de la optimización combinatorial a pesar de ser de propósito general. Una implementación a la medida, sacada de un paper académico, suele tener mejores resultados que una aplicación hecha sobre ILOG, pero la segunda es mucho más barata de producir y mucho más fácil de adaptar posteriormente.

Lamentablemente nuestro producto es de propósito específico por lo que es extremadamente frágil a cambios radicales en las restricciones. Para compensar en alguna medida esta debilidad se aislaron todos los cálculos de costos en una jerarquía de funciones objetivo de manera que si éstas cambiasen, los sitios a modificar estuviesen bien localizados. Estas funciones objetivo se pueden ver en el anexo

*Taburoute.cpp* como una familia de métodos consecutivos que tienen como prefijo $ObjetiveFunction()$. Adicionalmente se ha utilizado un sistema de etiquetas que marcan donde habría que agregar código en caso de añadir o eliminar alguna restricción. Una posible mejora sería aislar aún más las funciones objetivo en una colección de objetos relacionados entre sí que pertenecieran a los elementos que participan en el cálculo (rutas, conexiones, etc.). Estos objetos podrían tener algún tipo de interruptor que apagara o encendiera determinados costos para facilitar las pruebas y depuración del producto.

# 7 PLANIFICACIÓN Y COSTES

Éste proyecto no tuvo una planificación global concebida de antemano. A la fecha de mi incorporación a Goal Systems no se había firmado ningún contrato con fechas que definieran la planificación de los desarrollos. En su lugar se conocía de la demanda en el mercado de productos de este tipo y se quería desarrollar un núcleo de cálculo que luego fuera posible vender a varios clientes simplemente cambiando la interfaz de éste y modificando detalles menores.

Como becario se me dio la posibilidad de investigar a fondo el problema y darle el alcance que quisiera. Semanalmente tenía que reunirme con el gerente de desarrollos para reportar los avances de la semana y comentar la planificación para la semana entrante. No obstante, mirando hacia atrás, puedo hacer una reconstrucción de cuales fueron los tiempos y los costes aproximados asociados a cada fase del desarrollo. Yo, como becario, era el único coste que tenía éste proyecto, por lo que todos los costes se han calculado tomando como base 9 euros por hora.

La evolución del proyecto la vamos a dividir en las siguientes fases:

- **Fase 1:** Toma de requisitos y compresión del problema.
- **Fase 2:** Investigación del estado del arte.
- **Fase 3:** Diseño e implementación de la arquitectura e interfaz gráfica.
- **Fase 4:** Adaptación e implementación de Clarke and Wright y Branch and Bound.
- **Fase 5:** Adaptación e implementación de *Taburoute*.
- **Fase 6:** Puesta a prueba y depuración de errores.

La Tabla 7.1 muestra un diagrama de Gantt con el tiempo dedicado a cada una de estas tareas. La Tabla 7.2 especifica los aspectos más importantes de cada fase. Como se puede apreciar, el coste total del proyecto asciende a 4500 **€.**

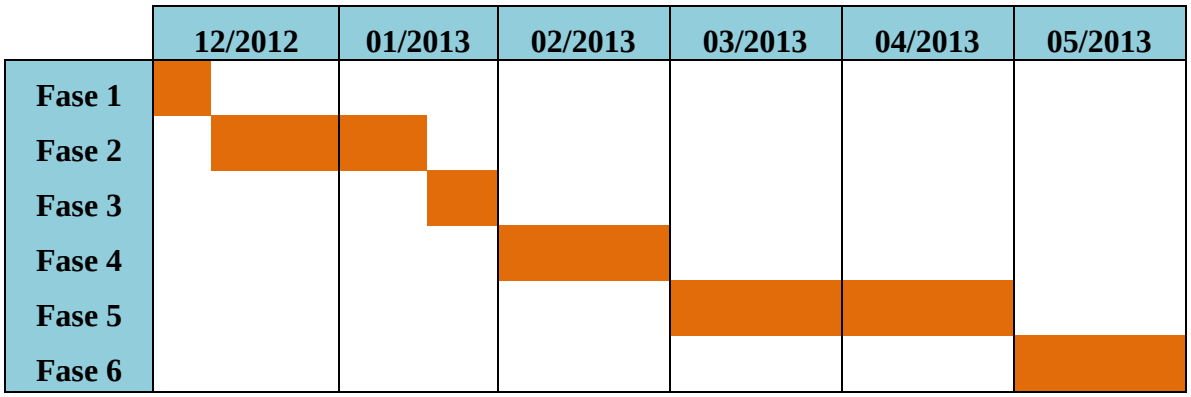


TABLA 7.1 DIAGRAMA DE GANTT CON LAS FASES DEL PROYECTO

| | Tiempo consumido | Cantidad de líneas de código | Coste aproximado |
|---|---|---|---|
| **Fase 1** | 1 semana | 0 | 180 € |
| **Fase 2** | 5 semanas | 0 | 900 € |
| **Fase 3** | 2 semanas | 1764 | 360 € |
| **Fase 4** | 4 semanas | 997 | 720 € |
| **Fase 5** | 9 semanas | 2720 | 1620 € |
| **Fase 6** | 4 semanas | 0 | 720 € |

TABLA 7.2 CARACTERÍSTICAS DE CADA FASE DEL PROYECTO

# 8 CONCLUSIONES

Durante la investigación realizada para este proyecto nos dimos cuenta de que la última generación de *meta-heurísticas* para el VRP constituye el estado del arte en cuanto a la inclusión de algoritmos en paquetes comerciales. Las mejores de éstas llegan incluso a encontrar soluciones óptimas para problemas de varios cientos de vértices. Aunque desde el año 2000 se han estado realizando propuestas novedosas, de momento *Tabu search* se mantiene como la técnica más efectiva. Aquellas técnicas basadas en Algoritmos genéticos o Redes de neuronas son claramente superadas, mientras que las basadas en *Simulated* o *Deterministic annaeling*, o Algoritmos de hormigas no ofrecen tiempos de cálculo competitivos. Un punto a observar es que no se han hecho suficientes estudios sobre el comportamiento de estas *meta-heurísticas* sobre instancias del problema mucho más grandes, que se encuentran con frecuencia en aplicaciones prácticas, en ocasiones llegando a tener decenas de miles de vértices. Dados los requerimientos computacionales, estas *meta-heurísticas* tan sofisticadas serán seguramente incapaces de resolver dichas instancias del problema en un tiempo de cálculo razonable. Es en estos casos cuando habría que recurrir a las heurísticas clásicas

y sacrificar calidad en las soluciones. Como promedio, las heurísticas clásicas ofrecen soluciones que distan entre un 2% y un 10% de la solución óptima (o mejor solución conocida en aquellos casos donde se desconoce la óptima), mientras que dicha cifra para las *meta-heurísticas* suele estar por debajo de 0,5%. La Ilustración 8.1 grafica lo planteado. En la última década los académicos han estado intentando desarrollar métodos capaces de proponer soluciones de buena calidad con poco tiempo de ejecución. Esto probablemente se logrará mejorando el tiempo de ejecución de las *meta-heurísticas* actuales en lugar de mejorar la calidad de la soluciones ofrecidas por las heurísticas clásicas. Un ejemplo de esta tendencia es el algoritmo de *Granular Tabu Search* propuesto por Toth y Vigo en el 2003 [10].

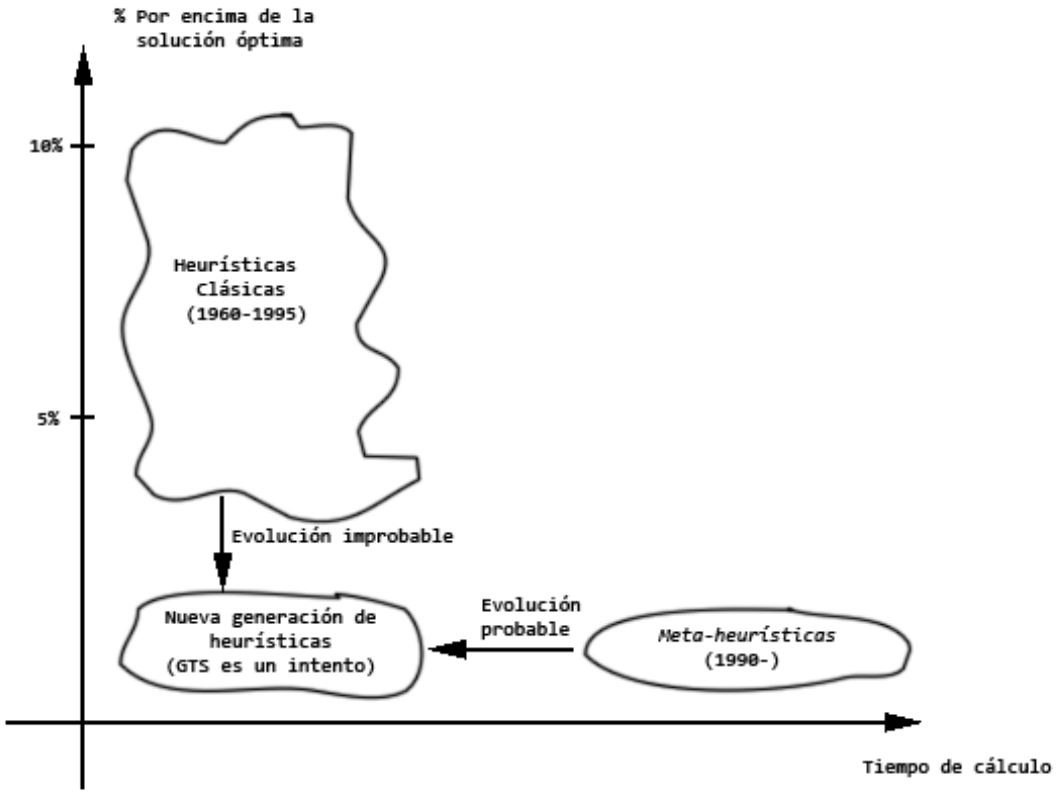


ILUSTRACIÓN 8.1 EVOLUCIÓN DE LAS HEURÍSTICAS PARA EL VRP

Otra conclusión sacada de este proyecto tiene que ver con la investigación científica en entornos empresariales, qué importancia se le suele dar y qué importancia creo que debería tener. Obviamente, las opiniones con respecto a este tema no son absolutas, sino que dependen en gran medida de la misión de la empresa que se esté analizando. No obstante, no sería arriesgado afirmar que en la mayoría de las empresas la investigación científica está estigmatizada y su uso es extremadamente limitado, o no existente. Incluso en una empresa de optimización como Goal Systems, no fue una tarea simple convencer a los gerentes de desarrollo de que 6 semanas de investigación pura tendrían resultados positivos a mediano plazo. Es triste ver como una y otra vez,

grandes equipos de ingenieros se pasan meses “reinventando la rueda” y lidiando con problemas complejos cuyas soluciones están escritas. No todas las empresas pueden permitirse invertir recursos en la investigación y para ser justos, muchas ni siquiera lo necesitan. Pero eliminar el estigma existente sería un primer paso importante para que la investigación científica eventualmente ocupe el lugar que merece en cada empresa.

# 9 BIBLIOGRAFÍA


[1] N. R. Achuthan y L. Caccetta, «Integer linear programming formulation for a vehicle routing problem,» *European Journal of Operation Research,* nº 1, pp. 86-89, 1991.

[2] G. Clarke y J. V. Wright, «Scheduling of vehicles from a central depot to a number of delivery points,» *Operations Research,* nº 12, pp. 568-581, 1964.

[3] D. Vigo, «VRPLIB: A vehicle routing problem instances library,» Università di Bologna, Bologna, 2000.

[4] A. Wren, Computers in Transport Planning and Operation, London: Ian Allan, 1971.

[5] A. Wren y A. Holliday, «Computing scheduling of vehicles from one or more depots to a numer of delivery points,» *Operational Research Quarterly,* nº 23, pp. 333-344, 1972.

[6] B. E. Gillet y L. R. Miller, «A heuristic algorithm for the vehicle dispatch problem,» *Operation Research,* nº 22, pp. 340-349, 1974.

[7] I. H. Osman, «Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem,» *Annals of Operations Research,* nº 41, pp. 421-451, 1993.

[8] M. Gendreau, A. Hertz y G. Laporte, «A tabu search heuristic for the vehicle routing problem,» *Management Science,* nº 40, pp. 1276-1290, 1994.

[9] M. Gendreau, A. Hertz y G. Laporte, «New insertion and postoptimization procedures for the traveling salesman problem,» *Operations Research,* nº 40, pp. 1086-1094, 1992.

[10] P. Toth y D. Vigo, «The granular tabu search and its application to the vehicle routing problem,» *Journal on Computing,* nº 15, pp. 333-346, 2003.

[11] B. L. Golden y A. A. Assad, Vehicle Routing: Methods and Studies, Amsterdam: North-Holland, 1988.

[12] G. Laporte, «The vehicle routing problem: An overview of exact and approximate algorithms,» *European Journal of Operational Research,* nº 59, pp. 345-358, 1992.

[13] M. Fischetti, P. Toth y D. Vigo, «A branch-and-bound algorithm for the capacitated vehicle routing problem on directed graphs,» *Operations Research,* nº 42, pp. 846-859, 1994.

[14] G. Laporte y Y. Nobert, «Exact algorithms for the vehicle routing problem,» *Annals of Discrete Mathematics,* nº 31, pp. 147-184, 1987.

[15] W. M. Garvin, H. W. Crandall, J. B. John y R. A. Spellman, «Applications of linear programming in the oil industry,» *Management Science,* nº 3, pp. 407-430, 1957.

[16] G. Barbarosoglu y D. Ozgur, «A tabu search algorithm for the vehicle routing problem,» *Computers and Operations Research,* nº 26, pp. 255-270, 1999.

[17] J. E. Beasley, «Route-first cluster-second methods for vehicle routing,» *Omega,* nº 6, pp. 154-160, 1983.

[18] B. Bullnheimer, R. F. Hartl y C. Strauss, «An improved ant system for the vehicle routing problem,» *Annals of Operations Research,* nº 89, pp. 319-328, 1999.

[19] L. Davis, Handbook of Genetic Algorithms, New York: Van Nostrand Reinhold, 1991.

[20] P. J. M. Van Laarhoven y E. H. L. Aarts, Simulated Annealing: Theory and Applications, Dordrecht: Reidel, 1987.

[21] J. Y. Potvin, «The traveling salesman problem: A neural network perspective,» *Journal on Computing,* nº 5, pp. 328-348, 1993.

[22] P. Toth y D. Vigo, The Vehicle Routing Problem, Philadelphia: SIAM, 2002.

[23] V. Pillac, M. Gendreau, G. Christelle y A. L. Medaglia, «A review of dynamic vehicle routing problems,» *European Journal of Operation Research,* vol. 225, nº 1, pp. 1-11, 2013.

[24] G. Desaulniers, J. Desrosiers y S. Spoorendonk, «The Vehicle Routing Problem with Time Windows: State-of-the-Art Exact Solution Methods,» de *Wiley Encyclopedia of Operations Research and Management Science*, New Jersey, John Wiley & Sons, Inc., 2010.